56. srečanje mladih raziskovalcev Slovenije 2022

PRIHODNOST JE ZAPISANA V QUBITIH

Raziskovalno področje: računalništvo ali telekomunikacije

II. gimnazija Maribor

Avtor: Aleksander Kalacun

Mentorja: Mitja Osojnik, Dr. Marko Jagodič

Maribor, april 2022

Kazalo

Kazalo slik

Seznam prilog

Povzetek naloge

S pomočjo simulacij kvantnih računalnikov in uporabe nekaterih eksperimentov, ki so narejeni na pravih kvantnih računalnikih, bomo podrobno raziskovali aktualne probleme, za katere kvantno računalništvo obeta spremembe v reševanju in razvoju ved, ki se s temi problemi ukvarjajo. Probleme, ki jih bomo reševali s kvantnimi računalniki, bomo primerjali tudi z reševanjem na klasičnem računalniku in raziskali, kako sta se oba pristopa odrezala v hitrosti in kakovosti reševanja problema. Izbor raziskovalnih problemov, ki jih bomo v raziskavi reševali, temelji na štirih področjih na katere bo imelo kvantno računalništvo največji vpliv. Rezultati in ugotovitve raziskovanja so skladni s svetovnimi raziskavami in ugotovitvami, vendar smo ugotovili tudi bila nekatera odstopanja, ki so bila nepričakovana. V primeru »strojnega učenja z matrico« je bil hibridni algoritem skoraj tako hiter in učinkovit kot, sicer ne najboljši, vendar pa še zmeraj delujoč, klasični. V nekaterih odstopanjih se je kvantna simulacija odrezala veliko slabše kot kvantni računalnik, predvsem v primeru potujočega trgovca je bil kvantni počasen in nenatančen. V naših eksperimentih je bil tudi v primeru Shorovega algoritma kvantni način veliko počasnejši, še posebej z večanjem težavnosti. Ugotovili smo, da so kvantne simulacije, ki so še daleč od učinkovitega kvantnega računalništva, sposobne reševati nekatere resnične probleme, ki imajo velik vpliv na razvoj znanstvenih disciplin, ki imajo velik pomen na naš vsakdan.

Research paper summary

With the help of quantum computer simulations and the use of some experiments done on real quantum computers, we will investigate in detail the current problems for which quantum computing promises changes in solving and developing the sciences that deal with these problems. We will compare the problems solving with quantum computer and with classical computer and explore how both approaches worked out regarding the speed and quality of problem solving. The selection of research problems that we will solving in the research is based on four areas in which quantum computing will have the greatest impact. The results and findings of the survey are consistent with global surveys and findings, but we also found some discrepancies that were unexpected. In the case of "machine learning with a matrix", the hybrid algorithm was almost as fast and efficient as, although not the best, but still working, the classic. In our research in the case of the Shore algorithm the quantum simulation was much slower, especially with increasing difficulty. We have found that quantum simulations, which are still far from effective quantum computing, are capable of solving problems that will have a major impact on the development of scientific disciplines that are of great importance to our daily lives.

1. UVOD

Kvantno računalništvo je zadnjem desetletju prebilo veliko mej in doseglo veliko ciljev, za katere so mnogi mislili, da so nedosegljivi. Kvantni računalniki predstavljajo naslednji velik izum, ki ga po vplivu na naš vsakdan mnogi enačijo s razvojem klasičnih računalnikov ali telefonov. Evropska Unija, vlade in največja tehnološka podjetja na svetu vlagajo v razvoj v tekmi, ki bo morda definirala naslednjo tehnološko revolucijo. Bliskovit razvoj in nove preboje vsako leto mnogi enačijo z računalniško revolucijo v 60-ih letih 20. stoletja. Takrat je bil koncept računalnika prav tako nepredstavljiv in le redki so videli potencial naprave, ki bi v prihodnosti spremenila svet, kot so ga računalniki. Od prvih konceptov Babbaga in Lovelace, ko so prvič sprevideli, da ima naprava, ki je sposobna računati potencial na veliko širšem področju kot računanje konkretnih enačb, do prvih delujočih modelov Turinga in Flowersa. Kljub temu je primerjava razvoja kvantnih računalnikov z razvojem klasičnih kompleksna. Že sam koncept ideje je drugačen. Pri razvoju klasičnih računalnikov je zaradi vojne prišlo do inovacije, ki je bila produkt obstoječih znanj matematikov in elektroinženirjev. Na razvoj kvantnih računalnikov pa je vplivalo odkritje lastnosti kvantne mehanike, ki je nato vodilo v ideje o aplikativni uporabi teh lastnosti. Kvantni računalniki so za razliko od klasičnih veliko bolj specializirani za točno določeno področje reševanja problemov. Tako da gre pri razvoju kvantnih računalnikov veliko bolj za nov koncept naprave, ki bi dopolnjevala pomanjkljivosti klasičnih, in sicer v smereh, ki imajo vpliv na nekatere resnične probleme. Vendar o končni uporabi kvantnih računalnikov danes še ne vemo veliko, tako kot tega niso vedeli začetniki razvoja klasičnih računalnikov. Eden izmed problemov, ki jih izpostavljajo skeptiki, glede kvantnih računalnikov, so tudi zakoni kvantne fizike, ki so nam zaenkrat dokaj neznani. Že danes se pri razvoju klasičnih procesorjev srečujemo s problemi, ki prihajajo s stalnim manjšanjem tranzistorjev. Ne vemo, ali bo fizično kdaj sploh mogoče imeti kvantni računalnik, ki bi imel več sto logičnih qubitov (kar bi danes pomenilo več sto tisoč fizičnih), že zaradi posebnosti pri novih zakonih kvantne mehanike. Zato še ne vemo, ali bomo sploh kdaj dobili zelo sposobne kvantne računalnike, in če jih, ali bodo res revolucionarni za naš vsakdan, ali bodo ostali le naprave za zelo specifične probleme, ki ne bodo imele veliko vpliva na svet. Edini način, da izvemo je, da poskusimo z razvojem kvantnih računalnikov, saj bi bilo neodgovorno in nespametno zapustiti razvoj verjetno največje inovacije na področju znanosti v zadnjih desetletjih, s potencialom spremeniti svet.

V nalogi se trudimo priti bliže k odgovorom na ta vprašanja. Preverili smo, kaj sploh so kvantni računalniki, kako delujejo in najpomembneje, želimo ugotoviti in videti, kako rešujejo probleme in kakšne so njihove aplikativne vrednosti za nas.

V začetnem, teoretičnem delu, bomo pobližje raziskovali zgodovino in razvoj. Pregledali bomo osnovne gradnike kvantnih računalnikov (kvantna vrata), kako ima vpliv kvantna fizika na delovanje kvantnih računalnikov, osnovne algoritme, ki to tehnologijo izkoriščajo in jih implementirali na kvantnem računalniku in kje so omejitve pri delovanju. Na koncu pa bomo to znanje uporabili in se lotili pragmatičnih problemov, v katerih bodo nekoč morda kvantni računalniki hitrejši od klasičnih. Z vsakega pomembnega aplikativnega področja bomo aplicirali nekaj problemov in zanje poiskali rešitev na kvantnem računalniku, nekaj problemov pa bomo rešili tudi na klasičnem in z aktualnimi metodami ter nato primerjali rezultate. Osredotočili se bomo na področja optimizacije, strojnega učenja, simulacij narave in kriptografije. Ugotovili bomo, ali si kvantni računalniki zaslužijo toliko pozornosti, ali bi pozornosti moralo biti še več, ali pa smo danes na prenizki stopnji razvoja, da bi lahko prišli do oprijemljivih zaključkov. Danes je bilo opravljenih že veliko eksperimentov na pravi kvantni strojni opremi, ki so jih promovirali kot “kvantno prevlado”, torej, ko kvantni računalnik v določeni aplikaciji prehiti klasični računalnik. Vendar pa je takšna definicija zelo kontroverzna in mnogi trdijo, da eksperimenti tega ne potrjujejo. Kvantno področje je še zmeraj zelo novo in razvijanje novih algoritmov in programske opreme za delo na kvantnih računalnikih se dopolnjuje in razvija na nekaj mesecev, medtem ko se število qubitov v novih kvantnih računalnikih podvojuje skoraj vsako leto.

Glede na navedeno, je glavni namen našega raziskovanja potrditi ali ovreči naslednje hipoteze:

Hipoteza 1: Kvantni računalniki oziroma simulacije so sposobni reševati resnične probleme v konkurenčnem času.

Hipoteza 2: Klasični računalniki imajo še zmeraj razvojno prednost, ki se zmanjšuje s hitrim razvojem kvantnih računalnikov.

Hipoteza 3: V kolikor algoritme kvantnih računalnikov simuliramo na klasičnih računalnikih, menim, da kvantni algoritmi ne morejo tekmovati z algoritmi, ki so prilagojeni samo klasičnim računalnikom - zato bodo klasične rešitve na klasičnih računalnikih zmeraj boljše od rešitev iz kvantnih simulacij na klasičnih računalnikih.

Hipoteza 4: Pri problemih optimizacije bo kvantna simulacija našla optimalno rešitev v precej daljšem času kot klasični računalnik, cena rešitve pa ne bo za veliko odstopala od optimalne.

Hipoteza 5: S stopnjevanjem problema optimizacije se bo čas večal do limita simulacije, natančnost oziroma cena pa bo padala s večanjem problema.

Hipoteza 6: Pri simulacijah narave bodo rezultati simulacije kvantnega računalnika večinoma v prednosti.

Hipoteza 7: V vseh primerih strojnega učenja bo kvantni računalnik počasnejši in zaradi hrupa manj natančen kot klasični. Razlika v natančnosti bo posebej velika pri večjih nevronskih mrežah.

Hipoteza 8: Pri uporabi Shorovega algoritma bo kvantna simulacija natančna, zaradi potrebnega procesa v ozadju pa bo trajala precej dlje kot na klasičnem. Z večanjem števila se bo čas eksponentno večal.

Kot glavno gradivo za teoretičen del je bil uporabljen učbenik Kvantno računalništvo in kriptografija Aleša Holobarja, v pomoč pa nam je bil tudi spletni učbenik Qiskit Textbook. Podrobneje o obeh virih v seznamu literature.

1.1 Metodologija dela

V nalogi raziskujemo potencial uporabe kvantnih računalnikov pri reševanju problemov, ki nas zanimajo in izhajajo s štirih področij na katere bo imelo kvantno računalništvo največji vpliv. Raziskovanje bo opravljeno pretežno preko simulatorjev kvantnih računalnikov na klasičnih, saj na žalost nimamo dostopa do najnovejših in najzmogljivejših kvantnih računalnikov na svetu. Kljub temu bodo nekateri enostavni algoritmi izvedeni na pravem kvantnem računalniku ponudnika IBMQ. Zaradi primerjave bomo nekatere probleme rešili tudi na klasičnem računalniku. Glede na to, da so kvantni računalniki v zelo zgodnji stopnji razvoja, smo za reševanje na klasičnem računalniku uporabili nam edino dostopen hardware. Vsi klasični problemi in simulacije kvantnih so bili izvedeni na procesorju AMD Ryzen 5 4800H, klasično strojno učenje na PyTorchu, pa je bilo opravljeno na grafični kartici Nvidia GeForce GTX 1650 Ti. Za uporabo sem uporabil odprtokoden software development kit (SDK) Qiskit, ki je bil izvajan na IDE Jupyter Notebook, vsi programi pa so bili izvajani na operacijskem sistemu

Linux distribucije Ubuntu. Za simulator je v večini primerov bil uporabljen splošen 32 qubitov obsežen qasm simulator, v nekaterih primerih pa je bil uporabljen State Vector simulator. Probleme smo prvo primerjali glede na težavnost. Za ohranjanje enostavnosti, je bil primerjan samo čas reševanje kvantnega in klasičnega računalnika, saj bi s težavo primerjali prostor na kvantnem in klasičnem računalniku.

2. RAZVOJ KVANTNIH RAČUNALNIKOV

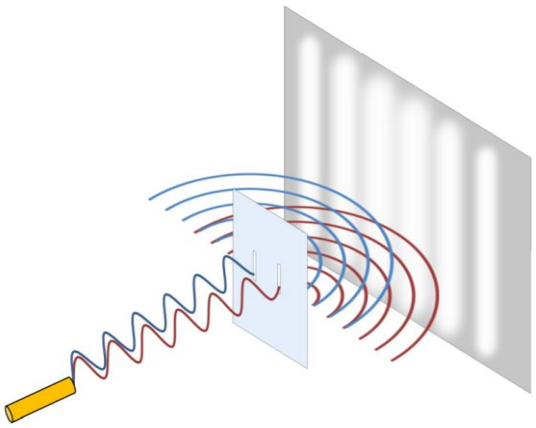
Kvantno računalništvo je kljub eksponentnemu razvoju v zadnjem desetletju, gledano z vidika časa razvoja novih tehnologij, kot idejna zasnova aktualna že dlje časa. Kljub že davno prepoznani uporabni in teoretični praktičnosti algoritmov, je ta tehnologija, zaradi kompleksnosti praktične uporabe, primerna šele zadnjih nekaj let. Za krajši opis razvoja same ideje moramo prvo povzeti razvoj vede kvantne mehanike. Precej fizikov je zagovarjalo različno stališče o dojemanju svetlobe, in sicer: Ali je svetloba sestavljena iz delcev (npr. Isaac Newton) ali gre za val? V odgovor je leta 1803 Thomas Young izvedel eksperiment dveh rež. Zaradi rezultata je v 19. stoletju, kljub nasprotujočim rezultatom drugih eksperimentov, svetloba veljala kot skupek valov. Šele Max Planck je leta 1900 z drugačno enačbo ovrgel teorijo. Vsekakor je bilo vprašanje o snovi svetlobe pomembna tema raziskovanj fizikov. Kljub začetnemu mišljenju o svetlobi kot valu, je na koncu obveljala teorija, da gre za delec in val verjetnosti. Radiacija toplote nam je iz fizike že znana. Vemo, da vsi predmeti s temperaturo višjo od absolutne ničle oddajajo toploto. V osnovi nastane zaradi povišane kinetične energije atomov in medatomskih trkov. Ti povzročijo pospešek naboja in dipolno nihanje, ki tvori elektromagnetno sevanje. Primer takšnega sevanja je tudi vidna svetloba, ki jo oddaja Sonce. Fiziki so prišli do teoretične ideje o predmetu, ki vsrka vso svetlobo in je ne odbija, prejeto energija pa seva v okolje. Predmet so poimenovali črno telo. Vsa energija, ki jo črno telo oddaja, ni enake jakosti. Jakost pa bi lahko izračunali na osnovi temperature s pomočjo Rayliegh-Jeansovega zakona. Problem pri izračunavanju je bil predvsem v tem, da so fiziki mislili na svetlobo kot val, kar je pomenilo, da je jakost povezana s frekvenco valovne dolžine svetlobe. Namesto premo sorazmerne frekvence in energije, je sevanje črnega telesa prišlo samo do neke točke ultravijolične svetlobe, potem pa je sevanje celo padlo. Eksperiment imenujemo »ultravijolična katastrofa«. Vedeli so, da ima elektromagnetizem omejitve v opisovanju svetlobe in energije. Tukaj se v zgodbo vključi nobelovec s področja fizike Max Planck. Revolucionarna ideja, ki jo je Planck predstavil leta 1901, je trdila, da je energija kvantizirana. Kljub Planckovem mnenju, da gre za protislovje v rezultatu, je bil Albert Einstein prvi, ki je ugotovil, da je svetloba kvantizirana. Zaradi omenjene teorije je Einstein dobil svojo prvo Nobelovo nagrado s področja fizike. Dognanje, da svetlobo sestavljajo fotoni, je v primeru eksperimenta z dvema režama veljalo za protislovno. Obveljalo je, da je foton delec in val verjetnosti hkrati, dokler ni moten z merilno napravo in s postopkom meritve. V trenutku

meritve se spremeni v delec in tvori obris dveh rež kot pričakovano. V nemerjenem eksperimentu, valovi med seboj delujejo kot običajni valovi med seboj, se izenačujejo in združujejo, mesto fotona na drugi strani rež pa je določeno kot verjetnost. Kvantizacija energije fotonov je nasprotovala takratnim zakonom fizike. Nobelovec s področja fizike Niels Bohr, je teoriji, po začetnemu neodobravanju, pripomogel h ključnemu razvoju teorije. Bohr je odkril, da med kroženjem elektronov, lahko zaradi spreminjanja energije v kvantih pride do kvantnega skoka med fiksnimi orbitalami. Kmalu je prišlo do veliko raziskovanj teorije, da je foton val in delec hkrati. To se je navezovalo na spoznanje, da ima vsaka snov svojo valovno dolžino, ki je obratno sorazmerna z maso telesa. Valovna dolžina človeka je zanemarljivo kratka, zato nima vpliva na obnašanje snovi kot celote, valovna dolžina ene molekule ali atoma pa je dovolj velika, da ima ta vpliv na obnašanje celotnega telesa. Tako se dejstvo, da je energija elektrona kvantizirana, navezuje na število valov v valovni dolžini elektrona, ki je odvisno od absorpcije energije drugih dolžin kot je foton. V matematično zakonitost je bilo to zapisano v Schrödingerjvi enačbi. Prišlo je do več idej o podobnem obnašanju materije, kar bi pomenilo dojemanje realnosti kot verjetnost. Leta 1927 je nobelovec s področja fizike Werner Heisenberg prišel do zaključka, da informacij o kvantnem delcu ne moremo vedeti s gotovostjo, saj ob boljšem poznavanju dolžine valova, manj vemo o obnašanju delcu samega. Bolj kot poznamo pozicijo delca, manj je znana njena gibalna količina, ki je odvisna od valovne dolžine vala verjetnosti. Približek izračuna obojega se računa s pomočjo primerjanja vrhov in dolin teh valov. Optimalno poznavanje obojega pomeni, da bo upoštevana tudi verjetnost. Ideji, da kvantna mehanika temelji na nedoločenosti, je nasprotoval Albert Einstein. Revolucionarna ideja, predvsem z vidika aplikativne uporabe kvantne mehanike, je kvantno prepletanje. Ideje, ki je Einstein ni mogel sprejeti, saj še vedno krši osnovne zakone fizike. Poenostavljeno, kvantni delci so med seboj povezani in sprememba lastnosti enega v istem trenutku vpliva na drugega. Lastnost je predvsem spin nekega kvantnega delca, ki takoj vpliva na drugega. Do danes še nerešljiv problem je predvsem v tem, da povezava med delcema deluje hitreje od hitrosti svetlobe oziroma v hipno in ne glede na razdalje v svetlobnih letih, kar v teoretični fiziki ni sprejemljivo oziroma je nerazumljivo. V vsebini naše naloge je naveden problem še posebej pomemben, saj je osnova za qubite. Tako kot običajni računalniški biti, ki delujejo na osnovi dveh stanj, qubiti lahko funkcionirajo kot 0 ali 1 stanje hkrati. Temu fenomenu prepisujemo sposobnost superpozicije. Pri superpoziciji gre za val možnosti, ki je rezultat motenj drugih valov možnosti med seboj (kot smo jih že prej omenjali v eksperimentu dveh rež). Kadar pride

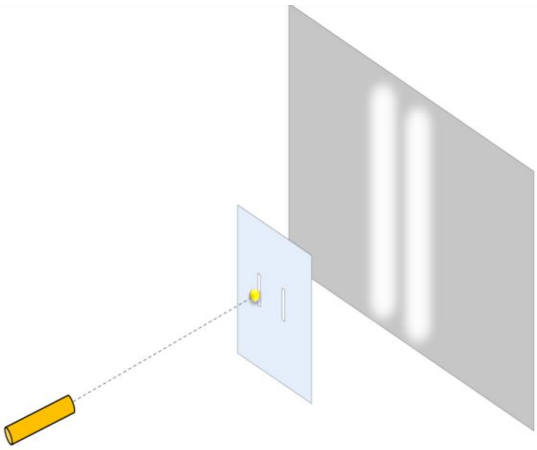
do prepletanje qubitov in do algoritmov, ki so to zmožni izkoristiti, dobimo kvantni računalnik. Trenutno je največji dosežek v količini 127 fizičnih qubitov, kar pomeni, da je že sposoben prehiteti običajen računalnik v določenih izračunih. Pri razvoju pa je problematična predvsem kvantna dekoherenca. Pogoj za uporabo qubitov je, da poznamo njihovo stanje, preden zaženemo algoritme za uporabo. Problem pri merjenju stanja qubitov je predvsem v zaznavanju vplivov drugih dejavnikov na qubite. Kompleksnejši sistem prepletajočih se qubitov imamo, težje je zaznati vse motnje, ki imajo potencialen vpliv na sistem. Takšni dejavniki so lahko tudi molekule in celo fotoni. Kot prej povedano, celo naša opazovanja vplivajo na sistem. Zdaj, ko smo spoznali, kako bi naj takšen računalnik v osnovi deloval, se lahko vprašamo, kako lahko te podatke pretvorimo (kot smo že prej spoznali, valov ne moremo meriti). Odgovor je, da mora val razpasti na informacije, ki jih lahko berejo klasični računalniki. To dosežemo s posebnimi algoritmi, ki pretvorijo te informacije v običajnemu računalniku razumljive.

Na sedanjem nivoju razvoja, kvantni računalniki predvidoma ne bodo veliko boljši od klasičnih pri reševanju veliko problemov in bodo celo počasnejši. Trenutno je glavna prednost uporabe kvantnega računalnika pri reševanju specifičnih problemov, kjer mora kvantni računalnik za rezultat izvesti veliko manj operacij kot klasičen računalnik. Rezultat tako dobimo veliko hitreje, kljub temu, da so te operacije počasnejše. Zaradi teh lastnosti, lahko sklepamo, da bo v prihodnosti imelo kvantno računalništvo velik vpliv na več znanstvenih ved. Za primer lahko vzamemo kemijo, kjer bi bila kompleksna simulacija molekul na atomski ravni za kvantni računalnik veliko hitrejša. Primer je simulacija molekule kofeina, za katero bi potrebovali 10^{48} bitov informacij, kar je preprosto preveč za praktično uporabo vsak dan. To bi vplivalo tudi na medicino, kjer bi lahko razumeli delovanje telesa (recimo možganov) na atomski ravni. S kvantnim računalnikom bi bila učinkovitejša izdelava gnojila, zmanjšali bi lahko CO_2 v ozračju z novimi spojinami, odkrili nove materiale za superprevodnike pri katerih ni potrebna zelo nizka temperatura. Za takšno odkritje bi bilo potrebno raziskovanje materialov na subatomski ravni, kjer bi kvantni računalniki bili veliko učinkovitejši od navadnih. Najpomembnejša naloga pa bi po moje bila strojno učenje in upravljanje podatkov. Napredno upravljanje velike količine podatkov in izdelava algoritmov v zelo kratkem časovnem intervalu bi imeli vpliv na napovedovanje dogodkov in načrtovanje objektov, ki bodo tolerirali odstopanja zunanjih dejavnikov. Še ena zelo pomembna aplikativna uporaba kvantne tehnologije je računalniška kriptografija, ki bo zaradi naprednih kvantnih računalnikov postala ogrožena. Kvantni računalniki bodo namesto preverjanja vsake možnosti posebej, lahko naenkrat pregledali vse

možnosti in kriptiranje se bo moralo s pomočjo kvantnih računalnikov spremeniti. Predvidevamo, da sta revolucija glede kriptografije in posledično morebiten zaton kriptovalut eni od slabih strani napredka kvantnih računalnikov. Algoritmi za izvajanje takšnih zahtev bodo morali biti naprednejši in že v osnovi drugačni od današnjih.



Slika 1: Prikaz eksperimenta z dvojno režo



Slika 2: Prikaz eksperimenta z dvojno režo. Slika povzeta po Kvantno računalništvo in kriptografija. Vir slik 1 in 2 je Kvantno računalništvo in kriptografija, (vir 1)

2.1 Schrödingerjeva enačba[1]

Je enačba, ki opisuje spreminjanje kvantnega sistema skozi čas z vidika lastnosti valovne dolžine in pozicije delca. Opisuje spreminjanje valovne funkcije in verjetnosti porazdelitve sistema. Splošna enačba je definirana kot $i\hbar\frac{\partial}{\partial t}\Psi(\boldsymbol{r},t) = \hat{H}\Psi(\boldsymbol{r},t)$. Kjer je h Planckova konstanta, $\frac{\partial}{\partial t}$

[1] Gradivo za poglavje sta (Encyclopaedia, 2021) citiran kot vir 3 in (HOLOBAR, 2016) - naveden kot vir 1.

pa nam pove parcialni odvod v času. Na drugi strani enačbe sta še hamiltonian sistema in $\Psi(\boldsymbol{r}, t)$ funkcije valovne dolžine in spreminjanje skozi prostor in čas. S to funkcijo računamo verjetnostno amplitudo za stanje ki ga bomo izmerili v nekem trenutku in prostoru.

3. OSNOVNE OPERACIJE KVANTNIH RAČUNALNIKOV [2]

Pregledali bomo vse zakonitosti in delovanje kvantnih računalnikov, tako da bomo s tem znanjem lahko pozneje sami sestavili in reševali probleme.

3.1 Kvantna vrata

Podobno kot klasični računalniki, imajo tudi kvantni določene vrste vrat, ki so osnova za računanje in se vežejo v kroge, kjer celotno delovanje (od CPE do spomina) poteka v osnovi na vratih, ki manipulirajo z izidom računa glede na prepletanje in postavitev vrat, ki temelji deluje na Booleovi logiki. Tako poznamo vrata kot so in, ali, če in samo če, itn. Podobno lahko velja za kvantne računalnike, le da imamo nekatera vrata, ki so specifična za kvantne, saj delujejo po zakonitostih kvantne fizike. Poglejmo si nekaj najosnovnejših vrat.

3.1.1 NE vrata

V kvantumu imamo dve vrednosti, ki sta si med seboj nasprotni. Skozi operacijo ne (not), se prva vrednost spremeni v drugo. Poznamo jih tudi v klasičnem računalništvu in so si sama nase inverzna.

3.1.2 Nadzorovana ne (C-NOT) vrat

V operacijo sta vstavljeni dve vrednosti, kjer je ena nadzorna vrednost, druga pa ciljna vrednost. Kontrolna vrednost vpliva na to, ali se ciljna vrednost spremeni v nasprotno vrednost. V primeru, da nadzorna vrednost ne ustreza vrednosti, ki je pogoj za izvršeno nasprotje, se ciljna vrednost ne spremeni. Pri vseh operacijah je pomembno zaporedje vstavljenih vrednosti (katera je ciljna in katera je nadzorna vrednost), vendar lahko operatorje obračamo po lastni želji. Morda se nam trenutne operacije posamezno zdijo dokaj neuporabne, vendar če so združene v premišljenem zaporedju, so uporabne za veliko koristnih izračunov. Skupne značilnosti najdemo lahko tudi z logičnimi vrati v navadnih računalnikih, ki so osnova za vse operacije, ki jih izvaja naš osebnih računalnik.

[2] Gradivo za vso poglavje sta učbenik Kvantno računalništvo in kriptografija - naveden kot vir 1. (HOLOBAR, 2016) in (Learn Quantum Computation Using Qiskit , 2020).

3.1.2 Hadamardova H vrata

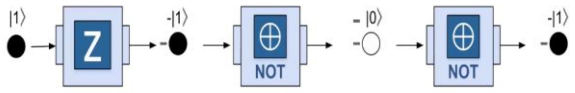
S primerom H vrat, ki spreminjajo qubite v stanje superpozicije, skozi katera vstavimo qubit neke vrednosti, rezultat skozi ena vrata pa je na prvi pogled povsem naključen. Ko za prva dodamo še ena vrata, ugotovimo, da je izpisna vrednost enaka vstavljeni. Ugotovimo, da je posledica tega superpozicija. Vizualno si lahko predstavljamo vrednosti kot ulomek, kjer pri drugi možnosti »fazo« označimo s znakom za negativno vrednost. Če vzamemo qubit v superpoziciji kot input v H vrata, ugotovimo da dobimo vrednost prvotnega qubita. Ker smo za stanje faze uporabili negativen znak, se nasprotni vrednosti izničita in vrednost skozi ponovno pretvorbo je enaka prvotni. Kadar je vrednost 1 v ulomku in verjetnosti qubita označena z negativnim znakom, je v fazi celoten qubit. Matematične operacije med qubiti in ulomki verjetnosti qubita lahko izvajamo kot z običajnimi ulomki. Za dodaten razvoj prikaza si lahko izmislimo še Z vrata, ki spremenijo ulomek verjetnosti qubita v fazo. Kadar pride do merjenja, superpozicija propade in stanje se odloča naključno.

Slika 3: Za pomoč si lahko predstavljamo vrednosti qubitov kot črne (1) in bele kroglice (0). Superpozicijo si lahko predstavljamo kot ulomek

Faza: V vizualni reprezentaciji zapišemo kot negativen znak na vrednosti 1, veljavna za ves qubit, če qubit ni v superpoziciji in je vrednosti 0, lahko damo negativen znak na vrednost 0.

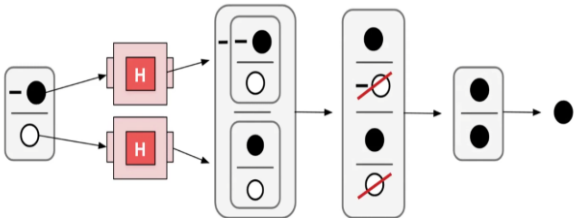
Ker superpozicijo predstavljamo kot ulomek in fazo kot negativno, lahko uporabljamo matematične zakonitosti, ki veljajo za te operacije.

Slika 4: Prikaz qubita s fazo v ulomku



Slika 5: Prikaz kroga računanja s kvantnimi vrati

Kot smo povedali, če qubit spustimo skozi dvojna zaporedna H vrata, dobimo vrednost, v kateri je bil qubit preden smo ga dali skozi H vrata.



Slika 6: Prikaz kroga s H vrati in razlike v fazi

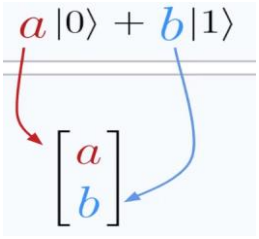
3.2 Notacija qubitov

3.2.1 Bra-ket notacija

Najbolj pogost način za zapisovanje qubitov je Bra-ket notacija. Ta nam pove verjetnost za obe vrednosti. Verjetnost merjenja prikažemo z Bra-ket notacijo. V Bra-ket notaciji nam kvadrat faktorja pred notacijo pove kolikšna je verjetnost za izmeritev te vrednosti v oklepaju. Če imamo dve vrednosti, je seštevek kvadratov faktorjev enak 1. Če velja, da je 1 v fazi, se v verjetnostnem računu napiše negativen namesto pozitivnega znaka ($\frac{1}{\sqrt{2}}|0|$ Kadar računamo s faktorji ki predstavljajo verjetnost se držimo pravil algebre. $a|0\rangle + b|1\rangle$

3.2.2 Vektorska notacija

Druga vrsta zapisa, ki je še bolj uporabna za kalkulacije med operacijskimi vrati, je vektorska notacija. Nenapisano pravilo pri pisanju Bra-ket notacije je, da je verjetnost, ki opisuje vrednost 0, vedno na začetku. Pomembno pravilo pri vektorski notaciji je, da se faktor vrednosti 0 piše na vrhu. Vektorski zapis predstavlja samo faktorje verjetnosti vrednosti, če pa gre za 0 ali 1 pa sklepamo iz pozicije v vektorski matrici. V vektorskem zapisu fazo označimo z negativnim znakom, kot pri Bra-ket notaciji. Negativni znak lahko tako kot faktorje izpostavljamo.



Slika 7: Računanje z matricami za en qubit.

Za računanje je bolj uporaben vektorski način, saj si ga lahko predstavljamo kot preglednico z elementi. Vsaka vrata (nasprotno, H, …) so predstavljena z matrico, preko katere lahko izračunavamo rezultat za določen qubit. Iz vektorske vrednosti lahko spremenimo v Bra-Ket notacijo, preko katere izračunamo verjetnost za dobljeno vrednost skozi vrata.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

Slika 8: Nazoren prikaz računanja matric in vektorjev zapisa qubitov

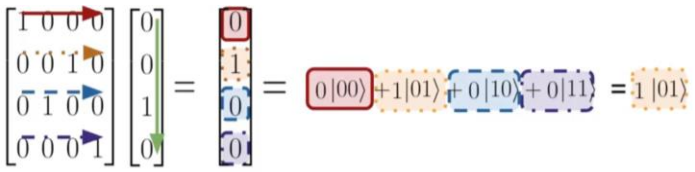
3.3 Kvantno prepletanje

Prišli smo do faze, kjer za razumevanje kompleksnejših sistemov kvantnih računalnikov potrebujemo osnovno znanje kvantnega prepletanja. Matematične operacije, ki smo jih že razložili, so enake kot v primeru za 1 qubit. Tudi nekaterih fizičnih zakonov, ki jih nismo znali razložiti (gravitacija, magnetizem, …), smo aplikativno uporabljali, dokler nismo odkrili splošnih zakonov, ki razlagajo vedenje teh sil. Po odkritju enačb, smo lahko fenomen uporabljali še bolj učinkovito in na več različnih načinov. Za ponovitev osnovnih dejstev iz uvoda: *kvantno prepletanje je pojav, kjer sta dva qubita (fotona, elektrona, ...) v superpoziciji povezana*. Ko pride do izmeritve enega izmed qubitov, ki je povezan v istem trenutku, ima vrednost v odvisnosti od drugega, ne glede na to, kako daleč narazen sta si ta povezana qubita. Prepletanje lahko deluje vzporedno (prvi ima po meritvi vrednost 1, zato ima tudi drugi enako vrednost), ali obratno (prvi ima vrednost 1 po meritvi, zato ima drugi vrednost 0). Kljub temu, da poznamo enačbe in lahko predvidevamo dogodke z izračuni, še vedno ne razumemo, zakaj je tako (podobno kot pri gravitaciji). Najpomembnejša aplikacija prepletenosti je, da po

izmeritvi enega izmed prepletenih qubitov, s 100% verjetnostjo vemo, kakšne vrednosti je drug qubit v paru, kar je ključno za računanje z več qubiti.

3.3.1 Računanje z več qubiti

Kadar imamo več qubitov v sistemu, lahko računamo kot prepletena ali ločljiva stanja. Kadar govorimo o ločljivih in kombiniranih stanjih, lahko sistem več qubitov ponazorimo s tenzorskim produktom, kjer računamo verjetnost določenih stanj za vsako stanje. Kadar imamo več qubitov (dve možnosti: 0 ali 1), računamo verjetnost rezultata za celotno skupino qubitov. Za izračun matrice vrednosti za več vzporednih qubitov, uporabimo operacijo tenzorjev produkt: $(\alpha|0\rangle + \beta|1\rangle) \otimes (\gamma|0\rangle + \delta|1\rangle) = \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle$ Nato lahko skupno stanje sistema vstavimo v matrico (vrata) in izračunamo stanje po množenju matric.



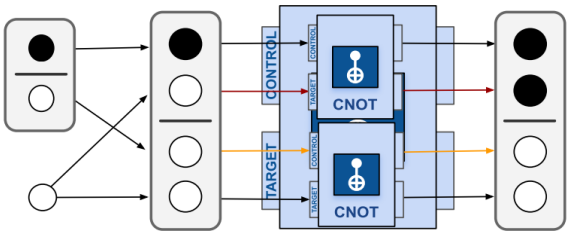
Slika 9: Nazoren praktičen prikaz množenja matric s qubiti

3.3.2 Računanje z več qubiti v superpoziciji

Držimo se pravila izpostavljanja:

1. Tenzorjev produkt (množenje v Bra-ket),
2. pretvorba v vektorski zapis,
3. izpostavljanja skupni faktorjev,
4. množenje skozi matrico.

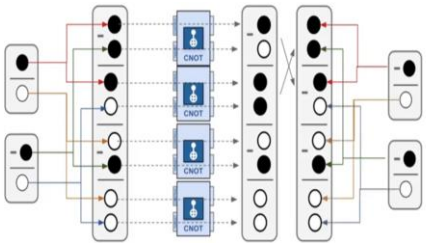
Reševanje z vizualnim prikazom:



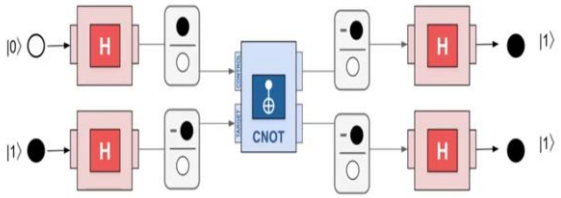
Slika 10: Za pomoč pri računanju je zraven dodan še vizualni prikaz

3.3.2.1 Faza

Pojasnjevali smo že fazo in kako pride do nje, če postavimo qubit vrednosti 1 v superpozicijo in kako jo ob računanju označimo z negativnim znakom. Vemo tudi, da faza velja za celoten qubit, vendar je pravilo, da če ima qubit možnost vrednosti 1, se označi za to vrednost. Če pa ima qubit vrednost 0, pa faze ni. Zamislimo si lahko nova, Z vrata, ki spreminjajo fazo. V matrici Z vrat (in vseh ki so izpeljanka iz Z) je vrednost -1.



Slika 11: Vizualni prikaz računanja vektorjev skozi matrice.



Slika 12: Prikaz računanja matric

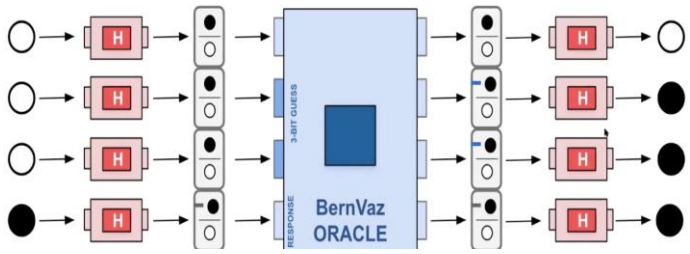
Zelo zanimiv primer je, če okoli nadzorovanih nasprotnih vrat postavimo H vrata. Kot vidimo, smo vrednost qubita 1 kopirali v vrednost 0. Temu pojavu pravimo “Kickback Phase” ali “posledična faza.”

3.4 Oracle algoritmi

So algoritmi, ki se zanašajo na to, da se z vsakim napačnim ugibanjem do rešitve algoritma poda uporabna povratna informacija, ki je uporabljena za reševanje problema. Oracle algoritmi so pomembni predvsem v varnosti. V osnovi računalnik ugiba kakšen vnos mora biti vstavljen, da bo povratna informacija najbolj uporabna k končni rešitvi.

3.4.1 Bernstein-Vaziranijev Oracle algoritem

Iščemo n-bitov dolgo kodo. S klasičnim računalnikom bi za to potrebovali n število poskusov. S pristopom surove sile bi jih potrebovali še več, saj bi morali preizkusiti vsako možno kombinacijo, vendar klasični računalniki uporabljajo IN vrata, kjer primerjajo binarno števko iskanega števila z naključnim številom. Tako z vsakim naključnim številom ugotavlja, ali je iskano število na tem mestu enako 1. Zato za n-bitov dolgo število potrebujemo n število poskusov. Kvantni računalniki so sposobni z Bernstein-Vaziranijevim algoritmom najti iskano število v enem poskusu. Sestavljen je namreč iz več C-NOT vrat, kjer je en bit reakcija na druga C-NOT vrata. Tudi če ne vemo, kateri qubiti so vezani na C-NOT vrata in imajo vpliv na četrti qubit, ki je v vlogi reakcije, lahko za n-dolgo kodo ugotovimo lokacijo v n-poskusih. Z različnimi vnosi sklepamo na katerem mestu so postavljena C-NOT vrata. To je zaporedni algoritem. Do bolj zanimivih rezultatov pride, če damo qubite v superpozicijo, oziroma uporabimo vzporedni algoritem.



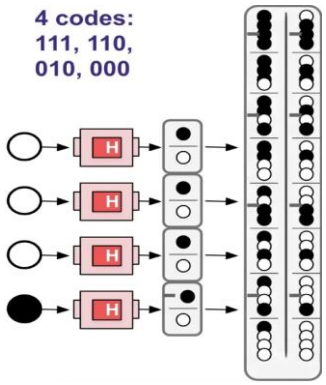
Slika 13: Vizualni prikaz kroga in reševanje s kvantnim računalnikom

V tem primeru do iskane vrednosti (011) pridemo že v prvem poskusu. Ker qubiti v superpoziciji držijo vse možne vrednosti v razmerju 50:50, po preobremenitvi skozi ponovna H-vrata dobimo iskano vrednost. Na vsaki točki imamo nadzorno vrednost 1 ali -1. Tako lahko ugotovimo, kje pride do sprememb v fazi po C-NOT vratih.

Implementirajmo algoritem še na kvantnem računalniku. Kot vidimo v **prilogi 1** je kvantni računalnik skrivno kodo oziroma zaporedje bitov res našel v prvem poskusu. Vidimo še prikaz kroga, kjer imamo, tako kot v opisu, v sredini razdeljena CNOT vrata, okoli pa H vrata, po katerih se na koncu kroga izmeri stanje qubitov in tako ugotovimo kodo.

3.4.2 Arhimed-Oracle algoritem

Tudi v tem primeru iščemo določeno 3-bitno skrivno kodo. V Arhimed-Oracle algoritmu se vrednost reakcijskega qubita spremeni v nasprotno vrednost. Predpostavimo, da želimo ugotoviti, ali imamo 4 skrivne kode, ali nobene. Če poskusimo z zaporednim algoritmom (poskušanje vsake kombinacije, dokler ne najdemo ene prave, oziroma 5 napačnih), bi potrebovali $\frac{n+1}{2}$ poskusov. Če poskusimo z vzporednim algoritmom, v primeru da je koda 0, v prvem poskusu ugotovimo, da se nič ne spremeni. V primeru da obstajajo 4 kode, naredimo kombinacije v superpoziciji iz vseh štirih qubitov.



Slika 14: Vizualni prikaz delovanja Arhimedovega oracle algoritma na kvantnem računalniku

3.5 Tenzorski produkt v računu matric

Vektorsko notacijo lahko tudi razširimo, tako da vsak index predstavlja vrednost v decimalnem sistemu, kadar imamo zapis qubita v binarni notaciji (01->1, 10->2, 11->3).

Računanje s tenzorskim produktom je nekomutativno, kar pomeni da je vrstni red računanja pomemben. Od leve proti desni ali od zgoraj navzdol. Včasih pišemo tudi brez vmesnega znaka.

$$|0\rangle \otimes |1\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix} = |01\rangle$$

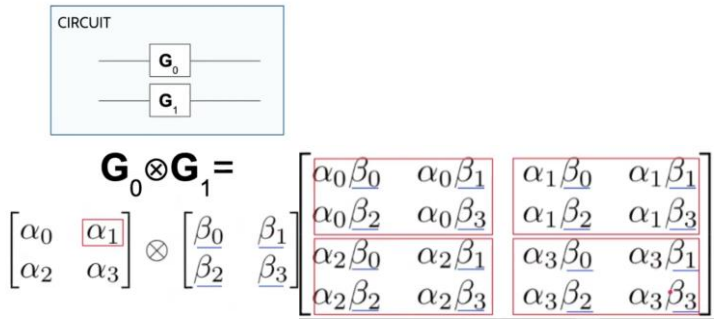
$$|1\rangle \otimes |0\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix} = |10\rangle$$

Produkte računanja v krogu lahko grupiramo skozi vrata, skozi katera poteka računanje qubita. Tukaj lahko predstavimo tudi identična vrata, ki stanja qubita ne spreminjajo in so enaka kot če

vrat ne bi bilo - stanje qubita se ohrani. Uporabna pa so pri grupiranju, ko dodajo matrico za lažje računanje tenzorskega produkta.

Ko bomo uprizarjali račune s tenzorskim produktom, bomo uporabili ⊗znak.

Ko grupiramo vrata za lažje računanje z več qubiti, če le ta niso namenjena za več vzporednih qubitov (vrata za zamenjavo), moramo izračunati skupno matrico skupine iz obeh vrat. To naredimo s načinom računanja tenzorskega produkta matric:



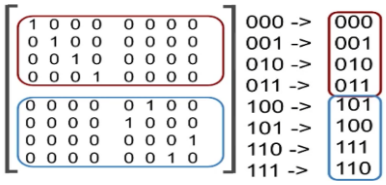
Slika 16: Prikazuje praktičen primer računanja dveh vrat, ki jih lahko na krogu združimo in računamo kot ena za vse vektorje

S tem sistemom lahko lažje računamo stanje več qubitov v vzporedni postavitvi vrat.

$$\begin{bmatrix}\alpha_0\\ \alpha_1\end{bmatrix}\otimes\begin{bmatrix}\beta_0\\ \beta_1\end{bmatrix}=\begin{bmatrix}\alpha_0\beta_0\\ \alpha_0\beta_1\\ \alpha_1\beta_0\\ \alpha_1\beta_1\end{bmatrix}\qquad \begin{bmatrix}\alpha_0 & \alpha_1\\ \alpha_2 & \alpha_3\end{bmatrix}\otimes\begin{bmatrix}\beta_0 & \beta_1\\ \beta_2 & \beta_3\end{bmatrix}=\begin{bmatrix}\alpha_0\beta_0 & \alpha_0\beta_1 & \alpha_1\beta_0 & \alpha_1\beta_1\\ \alpha_0\beta_2 & \alpha_0\beta_3 & \alpha_1\beta_2 & \alpha_1\beta_3\\ \alpha_2\beta_0 & \alpha_2\beta_1 & \alpha_3\beta_0 & \alpha_3\beta_1\\ \alpha_2\beta_2 & \alpha_2\beta_3 & \alpha_3\beta_2 & \alpha_3\beta_3\end{bmatrix}$$

3.5.1 Primer računanja matric v Bernstein-Vaziranijevem algoritmu

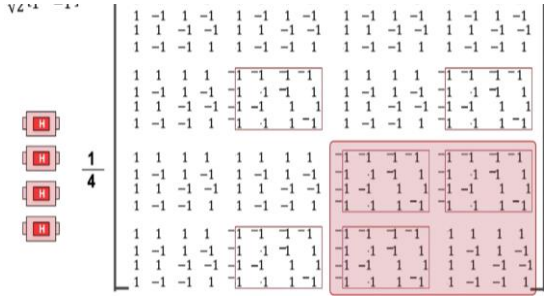
Primer računanja skupin vrat smo že omenili v Bernstein-Vaziranijevem Oracle algoritmu, kjer lahko izračunamo celotno skupino vrat za delovanje algoritma. Vemo, da za delovanje algoritma potrebujemo CNOT vrata, ki glede na vnos qubitov. Ti vplivajo na reakcijski qubit, ki spreminja stanje glede na postavitev CNOT vrat v krogu. Vsaka koda ima svoj unikaten krog. Glede na navedeno, lahko naredimo matrico za 3 qubite. Algoritem je 4 bitni, s tem da je 1 qubit reakcijski.



Slika 18: Matrica algoritma

Zdaj ko smo naredili matrico za zaporedno reševanje algoritma, poglejmo še za vzporedno.

Najprej dodamo H vrata, kjer naredimo matrico:



Slika 19: Matrica prikazuje združevanje 4 H vrat

3.5.2 Računanje produkta matric

Za računanje stanja nekega qubita, moramo množiti vektorski zapis qubita oz. skupine skozi neko matrico določenih vrat. Tega ne smemo mešati s tenzorskim produktom, ki združuje več vzporednih qubitov v enoten zapis in s tenzorskim produktom vrat, ki združuje matrice vrat v enotno skupino vrat. Množenje prav tako ni kumulativno.

Tako kot računamo rezultat nekega kroga, najprej združimo vzporedna vrata v skupne matrice s tenzorskim produktom, nato pa za končen rezultat izvedemo množenje matric vrat. Pri tem velja pravilo, da množimo z desne proti levi, glede na krog. Razlog za to je ta, da če želimo poznati stanje po drugih vratih, moramo najprej izračunati stanje po prvih. Za izračunavanje stanja med vrati, moramo poznati stanje pred zadnjimi vrati in temu stanju še dodati zadnja vrata.

3.5.3 Reverzibilnost

Problem pri merjenju popolne verjetnosti stanja qubita je odvisen od kvantne dekoherence. Za lažje razumevanje si lahko zamislimo škatlo, ki vsebuje polovico frnikol ene in polovico frnikol druge barve. Ker je škatla zaprta, je vsaka frnikola, dokler je ne vzamemo iz škatle in pogledamo, v stanju superpozicije. Če v škatli nastane luknja in iz škatle pade naključno število frnikol naključne barve, se verjetnost za dokončno stanje spremeni. Podobno je, ko pride do nepravilnosti v okolju qubita, kot sta nekonstantna temperatura ali magnetna sila.

Pogoj za kvantne kalkulacije je reverzibilnost. To pomeni, da se stanje qubita nikakor ne sme

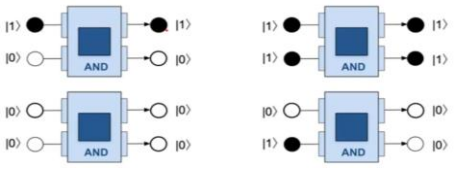
spreminjati, primer tega je dekoherenca. To pomeni, da je kakršen koli izračun, skozi matrico U , lahko obraten skozi matrico U^{-1} . Iz vsakega outputa lahko razberemo točno določen input.

$$|\psi_1\rangle = \mathbf{U}|\psi_0\rangle$$
$$|\psi_0\rangle = \mathbf{U}^{-1}|\psi_1\rangle$$

Nekatera vrata, ki smo jih do zdaj podrobneje pogledali (swap, not, …), so reverzibilna, hkrati pa tudi inverzibilna, saj stanje lahko obrnemo nazaj v prvotno z istimi vrati. V tem se kvantno računalništvo tudi razlikuje od klasičnega, saj klasični po navadi niso reverzibilni oz. reverzibilnost ni obvezna. Primer so IN vrata, saj je output 1 le, če sta oba inputa 1. Če je output 0, ne vemo kateri, če sploh katera, vrednost je bila 1. Ugotovimo lahko, ko je neka operacija reverzibilna, mora biti output edinstven glede na input, število vrednosti v outputu pa mora biti enako kot vrednosti v input. Vsak input naredi en output. Vendar kako naredimo vrata kot je IN v reverzibilna? Dodamo Ancillo, dodatna sredstva za izpolnjevanje pogoja reverzibilnosti. Upoštevanje kriterijev za reverzibilna vrata:

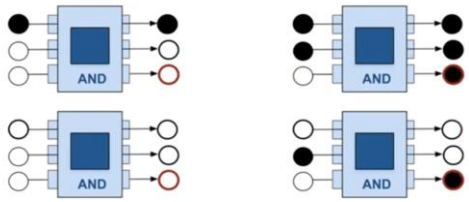
1. **število inputov je enako številu outputov.**

Da izpolnimo ta kriterij, dodamo še en output, tako da prenesemo enega izmed qubitov skozi, drugi pa nam pove stanje vrat.



Slika 20: Prikaz in vrat v pretvorbi

Ker trenutno stanje še vedno ne izpolnjuje drugih dveh kriterijev reverzibilnosti (vsak input mora določevati en edinstven output in vsak output mora biti določen od enega, edinstvenega inputa), bomo v naslednjem koraku dodali še en output, zgolj zato, da je končni rezultat specifičen za input.



Slika 21: Slika z Ancillo

Pri ugotavljanju outputa nam je v pomoč resničnostna tabela. S pomočjo Ancille lahko naredimo input/output kombinacije edinstvene, kar je pogoj za reverzibilnost.

3.5.4 Operacije s prepletenostjo

Poznamo dve vrsti prepletenosti: enako in nasprotno. Pri enaki prepletenosti se drugi qubit v paru spremeni v enako stanje kot prvi qubit, pri nasprotnem pa v nasprotno stanje. Kadar imamo v krogu več qubitov in različnih vrat, naredimo vektorski zapis za vse qubite, beremo od leve proti desni oz. od zgoraj navzdol, združimo vrata skupaj (tenzorski produkt pri vzporednih vratih in množenje pri zaporednih), od desno proti levi.

Nasprotno prepleten krog

Če vrata nimajo vzporednih za vse prepletene vzporedne qubite, uporabimo identična vrata.

Stanja, kot rezultat prepletenih krogov, so Bellova stanja.

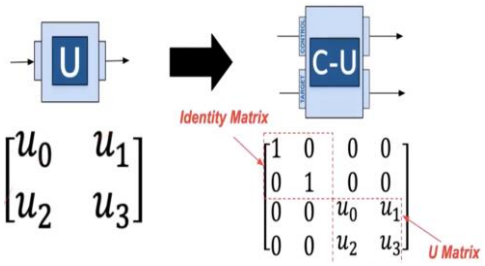
$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\1\end{bmatrix}, |\Psi^+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix}0\\1\\1\\0\end{bmatrix}$$

$$|\Phi^-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\-1\end{bmatrix}, |\Psi^-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix}0\\1\\-1\\0\end{bmatrix}$$

3.5.5 Kontrolna vrata

Pri preobračanju kontrolnih vrat (C-NOT in C-Z), se nekatere matrice spremenijo (c-not), nekatere pa ostanejo enake (c-z).

Katera koli vrata lahko spremenimo v kontrolna vrata, kot je C-NOT. Za tole uporabimo preprosto splošno matrico U.



Slika 22: Prenos C-NOT vrat v U matrico

V mesta označena z U lahko vstavimo katero koli matrico (primer: H vrata).

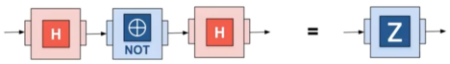
Če dodamo še eno kontrolo na c-not vrata, dobimo c-not ali Toffoli vrata, kjer se uporabljajo 3 qubiti, s tem da sta 2 v kontroli na tretjega, ki je ciljan qubit kot vidimo matrico velikosti 2^3.

Če pogledamo rezultate, vidimo da so outputi enaki inputom, razen ko sta prva dve vrednosti 1 in tretji 0, tretji spremeni vrednost v 1. Oba kontrolna qubita sta vrednosti 1.

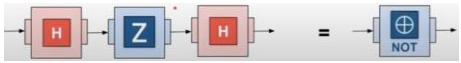
3.5.6 Pravila enakosti in komutativnost krogov

Set določenih vrat lahko deluje kot druga vrata ali seti drugačnih vratih.

Primer:



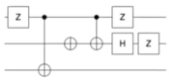
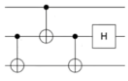
Slika 23: Prikaz komutativnosti kroga

in obratno 

Slika 24: Obraten prikaz vrat

To lahko dokažemo s množenjem matric vrat. Podobno pravilo velja tudi, če imamo namesto not c-not in namesto z c-z vrata. Ugotavljamo, da lahko vrata razstavljamo in potem spet združujemo, da dobimo nove enakosti med seboj. Kot smo že povedali, vrata v krogu med seboj niso komutativna, a obstajajo tudi izjeme.

Primer: okoli c-not vrat na ciljani strani lahko premikamo not in Z vrata. Poenostavitev kroga naredimo, ko zmanjšamo kompleksnost kroga, po navadi je to zmanjšanje števila vrat. Razveljavitev vrat pa nastane, kadar nastane identična operacija. To je pogosto, saj so vsa vrata reverzibilna. Navedli smo že, da to velja za zaporedni H vrati, vendar je to enako kot zaporedna

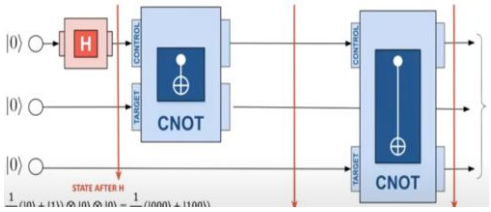
not in Z vrata, saj je rezultat identična operacija. Če pa imamo zaporedna c-not, c-z ali swap vrata, pa je rezultat tenzorski produkt identičnih vrat, kjer stanje ostane enako. Tako lahko 
poenostavimo v 
s pomočjo razstavljanja in razširitve. Temu pravimo dekompozicija kroga, ki je pomembna, saj lahko na kvantnih računalnikih uporabljamo omejeno število vrat.

3.5.7 Obsežna prepletenost in GHZ vrata

Kadar je v dveh qubitih verjetnost za obe stanji enaka in največja možna, rečemo, da je sistem maksimalno prepleten.

Primer: če imamo dva qubita enake prepletenosti, mora biti možnost za stanji 00 in 11 50%. Kljub temu, da so verjetnosti enake, je lahko 3-qubitno stanje ločeno.

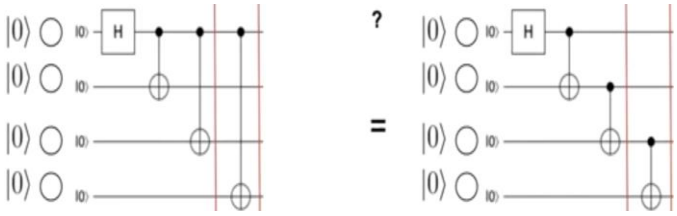
S pomočjo dodatnih c-not vrat, lahko vzpostavimo prepletenost med več kot dvema qubitoma.



Slika 25: Prikaz kroga s katerim naredimo prepletenost qubitov

Na začetku noben od qubitov ni prepleten. Po prvih c-not vratih pride do prepletenosti prvih dveh qubitov, saj je 50% možnost za 11 ali 00. Na koncu kroga je 50% možnost za 000 ali 111.

Spoznamo lahko Greenberger-Horne-Zeilinger stanja ali na kratko GHZ stanja. To so prepletena stanja, z vsaj tremi qubiti. To dosežemo z enimi H vrati in n-1 c-not vrati. Do zdaj smo postavili vrata na kontrolni del na prvi qubit. Kaj pa, če jih postavimo na naslednjega?



Po hitri kalkulaciji ugotovimo, da je output pri obeh postavitvah enak. Razlika med obema postavitvama je to, da je na pravem kvantnem računalniku težje vzpostaviti krog s postavitvijo

vrat v prvem primeru, zato je bolj uporabljen drugi. Primeri prepletenosti skozi GHZ vrata so primeri multipartitne prepletenosti. Uporabljamo pa jih lahko v komunikacijskih protokolih in kriptografiji.

3.5.8 Izrek o ne-kloniranju

Qubitov ne moremo kopirati. V klasičnem računalništvu se zanašamo na kopiranje pri računanju (v raznih algoritmih), pri shranjevanju (ko shranjujemo podatke v spomin) in pri odkrivanju napak (pri računanju in prenosu signalov). Poglejmo, kako bi izgledalo kopiranje qubitov:

$\mathbf{G}|\psi 0\rangle = \mathbf{G}(|\psi\rangle \otimes |0\rangle) = \mathbf{G}((\alpha|0\rangle + \beta|1\rangle) \otimes |0\rangle)$
$\mathbf{G}(\alpha|0\rangle \otimes |0\rangle + \beta|1\rangle \otimes |0\rangle)$ $\quad \alpha^2|00\rangle + \alpha\beta|01\rangle + \alpha\beta|10\rangle + \beta^2|11\rangle$
$\mathbf{G}(\alpha|00\rangle + \beta|10\rangle) = \mathbf{G}(\alpha|00\rangle) + \mathbf{G}(\beta|10\rangle)$ $\quad \alpha|00\rangle + \beta|11\rangle$

Nismo kopirali stanja, saj je druga vrednost stanje prepletenosti, kar pomeni, da bi ob merjenju ene vrednosti dobili vrednost druge. Informacije iz sistema so izgubljene. Iz tega lahko ugotovimo, da ne obstajajo vrata, ki bi omogočala kopiranje qubitov, saj se vedno izgubi določena informacija. Problem je, da ne moremo kopirati stanja v superpoziciji, saj ne moremo izmeriti (razpade ob meritvi).

3.5.9 Kvantna teleportacija

Kvantna teleportacija nam omogoča prenos qubitov v popolnoma ohranjenem stanju. Poteka lahko na velikih razdaljah. Seveda nam teleportacija ne omogoča potovanja s svetlobno hitrostjo ali prenosa snovi z dematerializacijo.

Za kvantno teleportacijo potrebujemo:

- 2 prepletena qubita,
- dodaten sporočevalni qubit, ki bo prenesen iz ene točke v drugo.
- klasično komunikacijo za prenos dveh bitov.

En prepleten qubit pošljemo v vsako točko (enega v A in drugega v B), sporočevalni qubit pa ostane v točki A. Tako izgleda skupno stanje:

$$\left|\frac{1}{\sqrt{2}}[\alpha|000\rangle + \alpha|011\rangle + \beta|100\rangle + \beta|111\rangle]\right.$$

Nato v točki A izvedemo c-not operacijo, kjer je ciljan qubit v prvotni prepletenosti, kontrolni pa v sporočevalni qubit. Spremenjeno stanje po prvi c-not operaciji:

$$\frac{1}{\sqrt{2}}[\alpha|000\rangle + \alpha|011\rangle + \beta|101\rangle + \beta|110\rangle]$$

Na sporočevalni qubit dodamo H vrata, ki spremenijo stanje na

$$\frac{1}{2}[\alpha|000\rangle + |011\rangle + |100\rangle + |111\rangle + \\ \beta(|001\rangle + |010\rangle - |101\rangle - |110\rangle)]$$

V zadnji točki izmerimo qubite v točki A.

00	01	10	11
$\alpha\|0\rangle + \beta\|1\rangle$	$\alpha\|1\rangle + \beta\|0\rangle$	$\alpha\|0\rangle - \beta\|1\rangle$	$\alpha\|1\rangle - \beta\|0\rangle$

Zgoraj: izmeritev v točki A

Spodaj: stanje v točki B po izmeritvi.

Nato pošljemo dva klasična bita z informacijami o merjenju iz točke A v točko B. V točki B po prejeti informacijami povrnemo ψ. Če je informacija 00, ni potrebno storiti nič, če je informacija 10, dodamo Z vrata, da spremenimo fazo, če pa je vrednost bitov 01, pa not vrata, da spremenimo verjetnost za obe vrednosti. Če pa je v točki A bila poslana vrednost 11, pa morata biti dodana obojna vrata. Po teleportaciji sta prepletena qubita uničena. Uporabljamo lahko proti računskim napakam, za oblikovanje kvantnih omrežij in za ultra varne komunikacijske kanale

Algoritem kvantne teleportacije sem prikazal tudi kot program na kvantnem računalniku v **prilogi 2**.

3.5.10 NISQ

Trenutno smo v NISQ (noisy intermediat scale quantum) obdobju razvoja kvantnega računalništva. Imamo med 50 in 150 qubitov, ampak so ti zelo šumeči, kar pomeni, da so zelo dovzetni za napake. Naš hardware pa je zelo omejen na velikost, na napake med računanjem in na koherenčne čase. Tehnik za popravo napak, pa še nismo odkrili, predvsem zaradi izreka o ne-kloniranju.

Ločimo na logične in fizične qubite. Logični so popolni qubiti brez napak in so uporabljeni za implementacijo v krogu. Fizični qubiti pa so fizično izpeljani na hardwareu (fotoni, ioni, itn.). Ti so zelo dovzetni za napake, za logičen qubit bomo potrebovali več kot 1000 fizičnih qubitov, in to šele, ko bomo odkrili tehnike za popravljanje napak. Logični qubiti so obvezni za nekatere algoritme.

Napake in njihovi vzroki:

- dekoherenca - pomeni, da imamo za kalkulacijo omejen čas, dokler stanje qubitov ne postane nestabilno, primeri so spreminjanje stanja (T1) in izničitev faze (T2),
- kalibracijska napaka,
- crosstalk,
- natančnost vrat,
- natančnost merjenja,
- inicializacija stanja.

Čeprav se napake zdijo majhne in nepomembne, se čez čas na velikih krogih kopičijo. Skozi več vrat pa se napake povečujejo, kar ima pomen za poenostavljanje vrat.

Oblikovanje kroga:

- povezanost qubitov med seboj za čim manj swap mrež, ki dodajo veliko časa,
- set vrat: kompleksna vrata morajo biti dekompozirana v osnovna, preprosta vrata za kvantni računalnik,
- software: čim bolj razvite in prilagojene compilerje in orodja za izkoriščanje vrat.

3.5.11 T in S vrata

Spoznajmo nova vrata, ki rotirajo fazo na Blochovi krogli (fi). Prva so S vrata $S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$. Druga pa so T vrata. $\begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix}$. (e je Eulerjeva konstanta (2,71828) osnova za naravni logaritem)

Lastnosti: $S^2 = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = Z$ pomeni, da sta dve zaporedni S vrati enski Z. Podobno velja za T, kjer so dvojna T enaka S vratom. $T^2 = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} = S$

Kako vrata vplivajo na rezultat za različne vrednosti qubitov?

Uporaba v kvantnem računalništvu

Med posegom različnih qubitov, je razlika samo pozitivna ali samo negativna in faza prevelika ali premajhna, kadar želimo računati stanja na bolj podrobnem nivoju. Ko je qubit v superpoziciji, kompleksne vrednosti veliko bolje opišejo neznano vrednost.

3.6 Groverjev iskalni algoritem

Groverjev algoritem je 1996 razvil Lov Grover. Je algoritem za iskanje v nestrukturiranem problemu iskanja, pri čemer kvadratno izboljša čas iskanja. Najprej definirajmo, kaj pomeni nestrukturirano iskanje. To pomeni, da v neki bazi podatkov iščemo predstavnika neke lastnosti, ki ima edinstvene lastnosti. Klasični računalnik mora v povprečju za to pogledati N/2 podatkovnih točk, v najslabšem primeru pa mora v bazi podatkov pogledati vseh N podatkovnih točk. S kvantnim računalnikom bi za rešitev potrebovali $\sqrt{N}$ poskusov. Napačno ugibanje ne pomaga pri iskanju končne rešitve - z napačnim ugibanjem ne najdemo zakonitosti, ki bi nam pomagale pri iskanju, za razliko od Oracle algoritmov.

Tako bi potrebovali O*N število poskusov (N je število vseh možnih rešitev). Če bi sortirali bazo in izpeljali binarno iskanje na urejenih podatkih, bi to skrajšalo čas na O(log(N)). Ampak sortiranje bi nam vzelo nekaj časa.

V osnovi Groverjev algoritem izkorišča superpozicijo in poseg faze iz O(N) na O($\sqrt{N}$). O je zgornja meja časa ali spomina, ki ga algoritem potrebuje.

Primer prejšnjega primera z Groverjevim algoritmom: $\sqrt{100}$=10.

Groverjev algoritem uporablja amplifikacijo amplitude. Tako povečamo verjetnost za opazovanje pravilne rešitve. Povečamo amplitudo povezano s ketom s pravim odgovorom in zmanjšamo amplitude za vse ostale možnosti.

Primer: če iščemo vrednost 11, bomo iz stanja

$$|\psi\rangle = \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle,$$

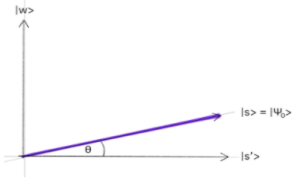
$$|\psi'\rangle = \frac{1}{\sqrt{12}}|00\rangle + \frac{1}{\sqrt{12}}|01\rangle + \frac{1}{\sqrt{12}}|10\rangle + \frac{\sqrt{3}}{2}|11\rangle$$

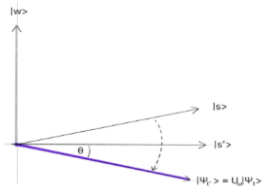
dobili stanje, kjer je bolj verjetno, da bomo dobili pravilen rezultat.

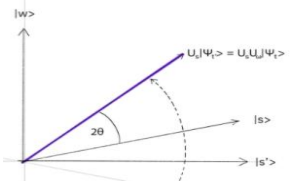
Potek: iz uravnoteženega stanja superpozicije izberemo vrednost in mu pripišemo fazo. To naredimo skozi $U_f$matrico. Recimo, da je lastnost, ki jo iščemo 101. Tako naredimo matrico:

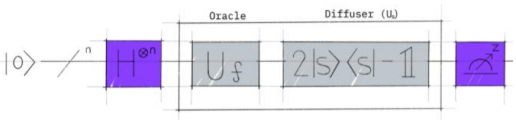
$$U_\omega = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \leftarrow \omega = 101$$

Groverjev algoritem je dober v problemih, kjer je rešitev enostavno preveriti, vendar težko najti, zato naredimo funkcijo f, ki je ob pravilni rešitvi enaka 0, ob napačni pa 1, kar pomeni, da je $U_\omega|x\rangle = (-1)^{f(x)}|x\rangle$ in matrica $U_\omega = \begin{bmatrix} (-1)^{f(0)} & 0 & \cdots & 0 \\ 0 & (-1)^{f(1)} & \cdots & 0 \\ \vdots & 0 & \ddots & \vdots \\ 0 & 0 & \cdots & (-1)^{f(2^n-1)} \end{bmatrix}$.

Ker ne vemo, kje je iskana podatkovna točka, lahko rečemo, da so vse lokacije enakovredne, kar lahko zapišemo kot uniformirana superpozicija $|s\rangle = \frac{1}{\sqrt{N}}\sum_{x=0}^{N-1}|x\rangle$. Tako lahko oblikujemo kompleksno ravnino 

, kjer še dodamo stanje s'. Tako lahko izvemo stanje $|s\rangle = \sin\theta|w\rangle + \cos\theta|s'\rangle$. V naslednjem koraku zrcalimo tako, da dobimo graf 

. Nato transformiramo graf v 

in z več poskusi prilagajamo rotacijo, dokler ne dobimo gotovega rezultata. Na koncu lahko narišemo Groverjev algoritem iz 3 delov: 

Slika 28: Enostaven prikaz kroga Groverjevega algoritma

Torej Groverjev algoritem je sestavljen iz dveh matric na n-1 qubitih. Prva je Oracle, druga pa difuzijski operator. Kako naredimo Oracle, ki ve kateremu ketu doda fazo?

Zamislimo si klasično funkcijo, ki ima vrednost 1, če je kriterij izpolnjen in vrednost 0, če ni. To je lahko iskanje optimalne poti, itn.

Nato naredimo reverzibilen krog, ki izračuna f(x) za vsak x. S pomočjo Ancille lahko naredimo reverzibilen krog iz ireverzibilnih krogov pri klasičnem računalništvu.

Dodamo Ancillo s fazo, da dobimo phase kickback, o katerem smo že govorili.

Potek celotnega algoritma:

- najdemo problem in podatke, ki nas zanimajo;
- najdemo najmanjši n, kjer je N$\leq 2^n$;
- damo vse qubite v superpozicijo;
- naredimo oracle in vrata U_f ;
- zaženemo algoritem skozi $U_f$in $U_{\square}$ $\sqrt{N}$ krat ;
- na koncu izmerimo stanja in dešifriramo stanja tistim ki ustrezajo v bazi podatkov.

Z Groverjevim algoritmom lahko iščemo tudi več vrednosti iz določene baze podatkov. Za t število iskanih vrednosti je potrebno $\sqrt{N/t}$ poskusov.

Algoritem smo implementirali na simulatorju za kvantni računalnik in ga izvedli na pravem kvantnem računalniku. Rezultate hitrosti reševanja smo med seboj težko primerjali, saj bi za reševanje na kvantnih računalnikih morale biti upoštevane čakalne vrste. Rezultati so v obeh primerih bili točni in za razliko od implementacije na klasičnem računalniku, je rešitev bila najdena v prvem poskusu. Primer programa s katerim smo izvajali Groverjev algoritem in primerjava s klasičnim računalnikom je v **prilogi 1**.

4. APLIKATIVNE SMERI KVANTNIH RAČUNALNIKOV [3]

Po odkritju klasičnih računalnikov, se še nismo zavedali vseh problemov, ki jih lahko računalnik reši. Vendar pa z razvojem novosti, kot je kvantni računalnik, pride do novih rešitev za obstoječe probleme, ki se jih verjetno še nismo zavedali ali pa nismo videli rešitve problema. Danes razvijamo algoritme in načine reševanja problemov za prihodnje zmogljive generacije kvantnih računalnikov in iščemo vedno nove rešitve za obstoječe probleme. Glede na trenutno poznavanje, lahko probleme razdelimo na probleme optimizacije, probleme simulacij narave, probleme strojnega učenja in probleme kriptografije. Sicer obstajajo tudi druga področja (npr. finančništvo), vendar so ta ozko povezana z eno izmed glavnih aplikativnih smeri problemov. V nadaljevanju naloge bomo z večih aplikativnih smeri raziskali nekaj problemov in jih rešili s programom za kvantni računalnik.

4.1 Računska zahtevnost

Gre za kategorizacijo problemov, glede na težavnost računanja na modelu Turingovega stroja. Merimo čas in prostor reševanja za računalnik. Za naše namene, ko bomo primerjali algoritme rešene na klasičnem in kvantnem računalniku, bomo bolj fokusirani na časovno karakterizacijo. V industriji kvantnih računalnikov je zadnja leta bilo predstavljeno merjenje zmogljivosti s kvantnim volumnom. To je meritev zmogljivosti in napak kvantnih računalnik, ki se meri v številu izvedenih naključnih kvantnih krogov. Vendar te metrike v nalogi nismo uporabljali, saj je primerjava v časovni uspešnosti bolj nazorna za naše potrebe. V računski zahtevnosti se sprašujemo, koliko težje je rešiti problem, ko se ta veča. Vse označeno s P (polinomno) je relativno enostavno rešiti na klasičnem računalniku, saj to pomeni, da je rešljivo v polinomskem času. Za vse ostalo pa še nimamo učinkovitega algoritma za reševanje problemov. Druga delitev je NP, kar pomeni, da rešitev lahko preverimo v polinomskem času, kar pa ne pomeni da je problem hkrati P (kot pri Groverjevem algoritmu). Za NP probleme poznamo nadrazreda NP-težke in NP-polne. Nadrazred NP problemov so sicer tudi VP problemi (verjetnostno polinomski). Problemi z oznako BGP so rešljivi za kvantni računalnik in v teh bodo kvantni

[3] Gradivo za raziskovalno delo v tem poglavju je bil učbenik Learn Quantum Computing Using Qiskit - naveden kot vir 2, in Qiskit: An Open-source Framework for Quantum Computing - naveden kot vir 5.

računalniki boljši v reševanju kot klasični. Kot primer poglejmo faktorizacijo števil. Kako težko je faktorizirati število z osmimi mesti in koliko težje je faktorizirati število z devetimi mesti. V našem primeru je eksponentno.

4.2 Kvantna optimizacija

Optimizacija pomeni najti najboljši način za reševanje problema glede na dane omejitve. To je lahko najkrajša pot, najhitrejši čas, najmanjša cena, najmanjši vpliv na okolje, itn. Zato pri problemih optimizacije določimo strošek (cost funkcijo), kjer določimo parametre in vrednosti oziroma pomembnosti dejavnikov. Če jo želimo maksimirati, funkcijo pomnožimo z negativnim številom in nato minimiziramo. Cost funkcija nam pove, kako ustrezna je rešitev, zato se vedno trudimo funkcijo manjšati. Optimizacijo si lahko predstavljamo kot funkcijo f(x), od katere moramo najti minimalno vrednost min f(x), tako da ima funkcija rešitev.

Probleme optimizacije delimo na diskretne in neprekinjene (continous).

1. **Diskretni problemi**

Primeri problemov: algoritmi za najkrajšo pot, algoritmi za ujemanje, max flow, min cut, itn. Velikokrat so diskretni problemi v kategoriji NP (če jih prisilimo, da je rezultat celo število ali binarno število). Primeri so problem potujočega trgovca, celo-številni linearni problemi, itn.

2. **Neprekinjeni problemi (continous)**

V teh problemih je rezultat v realnih številih, tako da so rezultati lahko neskončni. Primeri algoritmov so gradient descent, linearni programi, non-convex optimizacija. Algoritme, katerim lahko določimo obe lastnosti, rečemo hibridni algoritmi. Znan je primer max-cut.

Večina kvantnih algoritmov je eksponentno (kvadratno hitrejših). Problem pri večini algoritmov je strošek rešitve. Če si predstavljamo, je cena kvantnega algoritma C $\sqrt{n}$ pri tem je n iskalna baza. Klasični računalniki imajo ceno c krat n. Pomembno je, da je klasični c veliko manjši od kvantnega C pred faktorja. To lahko računamo z n> $(C/c)^2$. Danes je C do med 10^{10} in 10^{12} slabši, kar pomeni da mora n biti manjši od 10^{20}. V pred NISQ eri, optimizacijski problemi še niso realni. C se ocenjuje danes tako visoko zaradi dela klasičnega računalnika, ki izvaja operacije za povezovanje fizičnih qubitov v logične.

Za reševanje problemov uporabljamo algoritme z dokazano zmogljivostjo in hevristične algoritme. Med klasične z dokazano zmogljivostjo, štejemo množenje matric in Dijkstrov algoritem, kot kvantne pa Shorov in Groverjev algoritem. Hevristični klasični so simulirano ohlajanje in genetski algoritmi, kvantni hevristični pa so kvantno ohlajanje in QAOA.

Na popularnih problemih optimizacije bomo prikazali delovanje kvantnega računalnika za reševanje problemov. Pogledali bomo max-cut problem, problem potujočega trgovca in problem preusmerjanja vozil. Nato bomo prikazali razliko v klasičnih metodah reševanja problemov in kvantnem algoritmu. Naše mnenje (hipoteza) je, da bo kvantna simulacija pri vseh problemih optimizacije našla najboljšo rešitev, kljub temu, da bo časovno nekonkurenčna klasičnemu.

4.2.1 Max-cut

4.2.1.1 Reševanje s QAOA algoritmom

Je algoritem namenjen reševanju kombinatoričnih problemov z matrico U(β, γ), karakterizirano s parametroma za kvantno stanje $|\psi(\beta,\gamma)\rangle$. Najti moramo optimalna parametra, tako da kvantno stanje $|\psi(\beta opt,\gamma opt)\rangle$ pove rešitev. Problem lahko določimo z $U(\beta)=e-i\beta HB$, kjer je Hb hamiltonian mešanja (mixing hamiltonian) in Hp hamiltonian problema. Kvantno stanje lahko pripravimo tako, da matrico dodamo p-krat.

4.2.1.2 Utežen Max-Cut

Max-cut spada po zahtevnosti v kategorijo NP, kar pomeni, da je klasično zahteven. Spada med kombinatorne probleme. Kar pomeni, da izbiramo optimalen cilj na dan set. Zahtevnost se povečuje eksponentno. Max-cut je uporaben v clusteringu, omrežni znanosti, statični fiziki, marketingu, itn. V bistvu problema želimo razdeliti graf v dva podseta tako, da se največje možno število robov veže na točke v obeh podsetih. Vsaka točka ima uteži, ki prestavljajo neke podatke, lastnosti. Max-cut nam odgovori, kateri je optimalen izbor točk, da dosežemo nek rezultat. Za analogijo lahko vzamemo problem marketinga, kjer z grafom predstavljamo verjetnost nakupa produktov i ob morebitnih znižanjih, ki jih predstavlja j. Mi želimo najti povezavo med i in j, tako da bomo izbrali znižanje ob katerem bo največ ljudi kupilo produkt.

Matematičen zapis:

$$C(\mathbf{x}) = \sum_{i,j} w_{ij} x_i (1 - x_j) + \sum_i w_i x_i$$

kjer je $w_{ij} > 0, w_{ij} = w_{ji}$.

Cost funkcija v tem primeru je zmnožek uteži in povezav med različnimi podseti na vsaki strani razdelitve, w predstavlja verjetnost s katero bo kupec j kupil izdelek i.

Za kvantni računalnik moramo narediti ising hamiltonian, katerega formula je:

$H = \sum_i w_i Z_i + \sum_{i<j} w_{ij} Z_i Z_j$, kjer je Z Paulijev operator in ima eigen-vrednost +-1, $x_i \to (1 - Z_i)/2,$

Za mapiranje na kvantni računalnik pa uporabimo formulo:

$$C(\mathbf{Z}) = \sum_{i,j} \frac{w_{ij}}{4} (1 - Z_i)(1 + Z_j) + \sum_i \frac{w_i}{2} (1 - Z_i) = -\frac{1}{2} \left(\sum_{i<j} w_{ij} Z_i Z_j + \sum_i w_i Z_i \right) + \text{const}$$

Izvajanje algoritma:

1. Izberemo wi in wij.
2. Izberemo velikost našega kvantnega kroga.
3. Kontrolo θ, narejeno iz faznih vrat, rotacij Y in parametriziranih komponent.
4. Ocenimo $C(\boldsymbol{\theta}) = \langle \psi(\boldsymbol{\theta}) | H | \psi(\boldsymbol{\theta}) \rangle = \sum_i w_i \langle \psi(\boldsymbol{\theta}) | Z_i | \psi(\boldsymbol{\theta}) \rangle + \sum_{i<j} w_{ij} \langle \psi(\boldsymbol{\theta}) | Z_i Z_j | \psi(\boldsymbol{\theta}) \rangle$, kjer naredimo izid kroga in pričakovane vrednosti. Ocenimo točke okoli θ s klasičnim optimizatorjem.
5. Postopek ponavljamo dokler C(θ) ni najbliže θ.

v našem primeru bomo za poskusno funkcijo uporabili $|\psi(\theta)\rangle = [U_{\text{single}}(\boldsymbol{\theta}) U_{\text{entangler}}]^m |+\rangle$, kjer je $U_{\text{single}}(\theta) = \prod_{i=1}^{n} Y(\theta_i)$, oziroma skupek C-Faznih vrat, n število qubitov, m pa velikost kroga.

4.2.1.3 Formuliranje

Max-cut graf lahko formuliramo z Ising modelom. Graf označimo in dodamo na vsak rob utež, ki je lahko karkoli. Zdaj lahko vsaki točki dodelimo z binarno vrednost in jo dodamo v set S_+ ali S_- . Tako dodelimo vsaki točki v setu S_+ vrednost 1, vsaki v S_- pa -1. Tako lahko točke formuliramo v $\sum_{i,j=1}^{N} w_{ij}(1 - \tilde{z}_i \tilde{z}_j)$, kar pomeni, da bo produkt dveh točk v istem setu -1 in v različnem setu +1. Tako se izognemo prispevku teže dveh robov, ki sta v isti točki, ampak že v različnih

setih. Zdaj moramo le še maksimirati cost-funkcijo. To lahko klasično naredimo na veliko načinov. Seveda je en način, kako bi se to dalo narediti z izčrpnim iskanjem (exhaustive search), kar je učinkovito za manjše grafe kot je naš primer, vendar zelo težko izvedljivo na večji bazi. Drugi način bi bil razdelitev na sete naključno z Monte Carlo pristopom. Najbolje rešujemo to z sprostitveno tehniko, kjer rešimo enostavnejši temu podoben problem.

4.2.1.4 Reševanje s kvantnim računalnikom

Mapiranje algoritma na kvantni računalnik

Mapiranje se zanaša na kvantizacijo grafov. To naredimo tako, da točke preuredimo v operatorje - kvantne spine, ki vplivajo ena na drugo. Tako lahko robove predstavljamo kot interaktivne operatorje. Pretvorimo jih v energijska stanja. Zdaj se tega lahko lotimo s procesom kvantnega ohlajanja, ki bazira na VQE algoritmih. Na to lahko gledamo kot na minimizacijo energije. To lahko dosežemo, da med spini ni interakcij. Temu rečemo »easy to prepare« stanje. Tako sistem, kjer med spini ni interakcij, postopoma spreminjamo v sistem z vsemi interakcijami.

Drugi pristop k reševanju je z VQE algoritmom. V tem postopku pripravimo vnaprej parametrizirano stanje in izmerimo energijo. Nato spreminjamo parameter, tako da se spreminjajo proti najmanjši možni energiji sistema. Splošno pri optimizaciji imamo ciljno funkcijo, ki mora biti maksimalna ali minimalna.

Poleg opisanega procesa reševanja, smo dodali zraven še reševanje klasičnega računalnika po pristopu s surovo silo. To je najmanj zmogljiv način reševanja in obstaja veliko boljših načinov. Kljub temu je pričakovano veliko hitrejši od naše simulacije kvantnega računalnika na klasičnem. Simulacijo kvantnih računalnikov smo preizkušali še z več točkami. Pričakovano je klasično reševanje potrebovalo eksponentno več časa z večanjem števila mest, pri simulaciji kvantnega reševanja pa se je simulator ustavil pri iskanju rešitve za več kot 6 točk. Podrobnejša programa z grafi sta v **prilogah 4 in 4a**. Naša hipoteza na problemu potujočega trgovca velja, kljub temu, da je naša testna baza relativno majhna.

4.2.2 Problem potujočega trgovca

Je eden izmed precej znanih računalniških problemov, ki obstaja že dolgo med nami (začetki v 19 stoletju). Pri problemu je potrebno najti najboljšo pot med določenim številom destinacij. Spada v kategorijo NP-težkih Pri malo različnih destinacijah je možnosti znosno malo (3 destinacije, 6 možnosti), vendar rastejo možnosti glede na število destinacij fakultetno (n!), kjer že pri sedmih različnih destinacijah obstaja 5,040 različnih možnosti. Pri še večjem številu destinacij imajo današnji računalniki probleme z optimalnim izračunom, predvsem zaradi načina shranjevanja informacij. Sicer za to obstaja več znanih algoritmov.

V problemu rešujemo najkrajšo pot med mesti za potujočega trgovca, ki prodaja blago po različnih mestih. Trgovec se mora najhitreje vrniti v svoje začetno mesto. Problem predstavimo kot graf, kjer moramo najti najkrajši Hamiltonov cikel med vsemi točkami. To je pot, ki povezuje vse točke na grafu in uporabi vsako pot samo enkrat. Na grafu: $G = (V, E)$ s točkami $n = |V|$ in dolžinami w_{ij}. Graf lahko opišemo z $x_{i,p}$ spremenljivkami, ki jih je N^2 , kjer i predstavlja točko in p njeno mesto v ciklu. To pomeni, da se vsaka točka lahko pojavi samo enkrat v ciklu in da se na vsaki stopnji v ciklu mora pojaviti vsaj ena točka. $\sum_i x_{i,p} = 1 \ \forall p$ in $\sum_p x_{i,p} = 1 \ \forall i$.

Zato če sta $x_{i,p}$ in $x_{j,p+1}$ 1, se doda kazen $\sum_{i,j \notin E} \sum_p x_{i,p} x_{j,p+1} > 0$ energijska kazen in minimizirano enačbo lahko zapišemo kot $C(\mathbf{x}) = \sum_{i,j} w_{ij} \sum_p x_{i,p} x_{j,p+1}$. To sestavimo v splošno enačbo:

$C(\mathbf{x}) = \sum_{i,j} w_{ij} \sum_p x_{i,p} x_{j,p+1} + A \sum_p \left(1 - \sum_i x_{i,p}\right)^2 + A \sum_i \left(1 - \sum_p x_{i,p}\right)^2$, prosti parameter A pa določamo s $A > \max(w_{ij})$. Naš problem ponovno mapiramo v kvantni računalnik in določimo minimalni Ising Hamiltonian. Zdaj bomo problem rešili na klasičnem računalniku, z več pristopi in na kvantnem, rezultate pa bomo primerjali med seboj po času reševanja in pravilnosti rešitve.

4.2.2.1 Klasični pristopi

1. **Surova sila (Brute-force)**

   V pristopu s surovo silo (brute-force) računalnik preizkusi vsako kombinacijo poti posebej in dolžino poti nato primerja z vsako drugo kombinacijo posebej. Je najmanj primeren način za reševanje tovrstnega problema, kot vidimo na primeru.

2. **Dinamično programiranje**

   Izberemo si eno vozlišče, ki je začetna in končna točka hkrati. Nato za vsako drugo vozlišče i iščemo najmanjšo cenovno funkcijo za i kot končno točko na cilju. Cena je odvisna od dolžine v ciklu, izračunamo pa jo z razdeljevanjem cikla v podsete. Te podsete lahko obravnavamo posamezno kot sub-probleme. Za sub-probleme izračunamo ceno.

3. **Simulirano ohlajanje (simulated annealing)**

   Za primerjavo smo vzeli še rešitev iz pristopa simuliranega žarjenja, vendar ga ne bomo navajali kot primerno primerjavo.

4.2.1.2 Rezultati

Za naš preizkus bomo uporabili QASM simulator na kvantnem računalniku, preizkusili pa bomo delovanje na 4 mestih. Za boljše razumevanje se bomo osredotočali predvsem na primerjavo med kvantnim reševanjem in načinom s surovo silo. Rezultati dinamičnega programiranja so prav tako podani poleg. Zraven bomo merili čas in natančnost. Kot vidimo v eksperimentu, je kvantni pristop našel drugo najboljšo pot, medtem, ko je za primerjavo, reševanje s surovo silo našlo najboljšo možno pot. Rezultat, da kvantni na testiranju za več kot 3 mesta ni mogel najti najboljše poti nas je presenetil. Čas prav tako govori, da je trenuten klasični pristop s surovo silo hitrejši. V testiranju smo želeli še povečati število mest, vendar je naša testna oprema za kvantni simulator po 20 minutah odpovedala. Pri testiranju drugih klasičnih načinov sta pričakovano dinamično programiranje in simulirano žarjenje bila veliko hitrejša. Kljub temu je kvantni računalnik uspel rešiti primer s 3 mesti hitro in učinkovito, za današnje razmere pa je pri tako zahtevnem primeru preveč karkoli več kot 4 mesta. Programi in grafi z rezultati so podani v **prilogah 5, 5a in 5b**, reševanja s simuliranim žarjenjem se nismo odločili vključiti, rezultati pa so bili zelo dobri. Tukaj lahko našo hipotezo ovržemo, saj

simulacija kvantnega računalnika ni našla optimalne rešitve, navkljub veliko daljšemu času. To lahko pripišemo nepopolni simulaciji ali neučinkovitemu programu. Najverjetneje ima simulacija trenutno preveč hrupa za popolno delovanje.

4.2.3 Problem preusmerjanja vozil

Vsak dan se srečujemo z logistiko velikega števila vozil (tovornjakov, ladij, …) in postojank ali depojev (prenočišča, prevzemne lokacije, itn.). Glavni izziv je zasnovati poti med depoji in lokacijami, kamor so vozila namenjena, tako da sta pot in čas najkrajša možna.

Za problem potrebujemo:

1. Ciljne lokacije za vozila: v resničnem svetu so te že dane, vendar jih bomo mi generirali naključno.
2. Določiti razdalje med točkami in potovalni čas. Mi se bomo držali Evklidove razdalje.
3. Izračunamo poti.

Problem preusmerjanja vozil je kombinatoren problem, kjer iščemo najkrajše poti med depojem, lokacijami in potjo nazaj do depoja za določeno število vozil.

Naj bo število lokacij (kupcev) n, število vozil pa K. Naj bo x_{ij} = {0, 1} binarna spremenljivka, ki v stanju 1 aktivira segment od vozlišča i do j. Index je med 0 in n, kjer je 0 depo. Imamo dvakrat toliko spremenljivk kot je črt na grafu. Za popolnoma povezan graf n(n+1) binarnih spremenljivk. Če imata i in j povezavo med seboj, označimo $i \sim j$, z $\delta(i)^+$ pa označimo vsa vozlišča s povezavo do i. Za vsa vozlišča pa označimo tudi spremenljivke od 1 do n. Označimo z u_i. Tako lahko naš problem zapišemo kot: $(VRP) \quad f = \min_{\{x_{ij}\}_{i\sim j}\in\{0,1\},\{u_i\}_{i=1,\dots n}\in\mathbb{R}} \sum_{i\sim j} w_{ij}x_{ij}$.

Omejitve pri obiskovanju vozlišč pa $\sum_{j\in\delta(i)^+} x_{ij} = 1, \sum_{j\in\delta(i)^-} x_{ji} = 1, \forall i \in \{1,\dots,n\}$, kar zapove samo eno povezavo med lokacijami. Za obiskovanje depojev pa: $\sum_{i\in\delta(0)^+} x_{0i} = K, \sum_{j\in\delta(0)^+} x_{j0} = K$, ki pove da je lahko od in do depoja samo K povezav.

Stroškovno funkcijo določimo z razdaljo med i in j. Zraven določimo še omejitve, ki določajo, da je med dvema postojankama lahko samo ena povezava, druga da je lahko do vsake postojanke največ K povezav

4.2.3.1 Klasična rešitev

Za ta problem uporabljamo CPLEX, ki uporablja branch-and-bound-and-cut metodo za ocenjevanje rešitve. V tem primeru je to mešani celoštevilčni linearni program. V vektorju to zapišemo kot: $\mathbf{z} = [x_{01}, x_{02}, \ldots, x_{10}, x_{12}, \ldots, x_{n(n-1)}]^T$, kjer je $\mathbf{z} \in \{0,1\}^N$ in $N = n(n+1)$ Problem se povečuje kvadratno s številom vozlišč.

4.2.3.2 Kvantna rešitev

Uporabljamo klasični in kvantni del s VQE algoritmom.

Delamo v korakih:

- kombinatoren problem transformiramo v binarni optimizacijski problem;
- mapiramo problem v Ising hamiltonian glede na spremenljivke z iz baze Z z metodami kaznovanja;
- določimo velikost kvantnega kroga s m, velikost se prilagaja;
- določimo kontrole θ in s poskusno funkcijo $|\psi(\boldsymbol{\theta})\rangle$ naredimo krog C-faznih vrat in Y rotacij parametriziranih s θ ;
- ocenimo $C(\boldsymbol{\theta}) = \langle \psi(\boldsymbol{\theta}) \; H \; \psi(\boldsymbol{\theta}) \rangle$ z rezultatom kroga iz baze Z in dodajanjem vrednosti k posameznem hamiltonianu, tako določimo točke θ odvisne od optimizatorja;
- uporabimo klasični optimizator za set nadzornih točk;
- še naprej ocenjujemo C do minimalne vrednosti;
- na koncu še določimo končne parametre θ, iz katere naredimo zadnji set $\langle z_i \; \psi(\boldsymbol{\theta}) \rangle^2 \; \forall i$.

Na koncu predstavimo rezultate, grafi prikazujejo depo s zvezdico in vozila s puščicami, pot pa je povezana. Kot vidimo sta kvantni in klasični računalnik dobila različne rešitve, cena pri klasičnem računalniku se pri spreminjanju števila vozil in depojev ne spreminja veliko, medtem ko se cena pri kvantnem računalniku z večanjem števila mest in vozil veča. Ugotovimo lahko, da na primerih z veliko mesti in vozil klasični še vedno boljši in hitrejši, vendar so kvantni na simulatorju ta problem sposobni rešiti. Poleg smo testirali še več različnih primerov števila

vozil in depojev. Pri 4 vozilih in 6 depojih smo zadeli mejo 32 qubitov, ki jih naš simulator največ premore. Ugotovili smo, da stroškovna funkcija pri klasičnem reševanju veliko niha in ni odvisna od števila vozil ali depojev. Rezultati in program, ki je priložen v **prilogi 6**, je bil povzet po spletni strani:

https://qiskit.org/documentation/optimization/tutorials/07_examples_vehicle_routing.html

Rezultat kvantnega računalnika se prav tako razlikuje od rezultata klasičnega, za katerega lahko z gotovostjo trdimo, da je našel optimalno rešitev. Razlika v strošku med optimalno potjo in tisto s kvantno simulacijo, pa se z večanjem števila mest veča.

4.3 Kvantne simulacije narave

Matthias Troyer je rekel: "Če želimo shraniti vse možne konfiguracije za 125 orbital od neke kemijske spojine, bi potrebovali več klasičnega spomina kot je atomov v vesolju." S kvantnim računalnikom bi potrebovali le 255 qubitov. Teh še zaenkrat nimamo, vendar je v prihodnjih letih to dosegljiva številka.

Kvantne simulacije nam omogočajo raziskovanje s področja fizike kondenzirane snovi, veje fizike, ki preučuje fizikalne lastnosti kapljevin in trdnin. Z odkrivanjem lastnosti polprevodnikov je prišlo do digitalne revolucije. Torej fizika odkrivanja novih materialov, na katere lahko gledamo kot na določene konfiguracije atomov z določenimi fizikalnimi lastnostmi. Da nek material doseže fizikalne lastnosti kot so električne, magnetne, itn., je potrebno najti konfiguracijo atomov, ki najbolje ustrezajo našim potrebam za nov material. Primer takšnega materiala so superprevodniki. Najboljši superprevodniki danes potrebujejo nepraktične pogoje kot sta zelo nizka temperatura in tlak. S kvantnimi simulacijami lahko vse atomske konfiguracije preiščemo in preizkusimo veliko hitreje, kot če bi materiale vsakega posebej izdelovali in preizkušali.

4.3.1 Hamiltonian

V simulacijah bomo veliko uporabljali hamiltonian ali lastnosti, ki jih moramo upoštevati pri izračunavanju. Molekulo lahko predstavljamo kot skupek atomskih jeder in v orbitalah okoli letečih elektronov. Atomi so skupaj združeni v molekulo. S Hamiltonianom predstavljamo skupno energijo teh elektronov in jeder v sistemu. To lahko predstavimo z eno enačbo:

$$= -\sum_{i=1}^{N} \frac{1}{2}\nabla_i^2 - \sum_{A=1}^{M} \frac{1}{2M_A}\nabla_A^2 - \sum_{i=1}^{N}\sum_{A=1}^{M} \frac{Z_A}{r_{iA}} + \sum_{j>i} \frac{1}{r_{ij}} + \sum_{B>A} \frac{Z_A Z_B}{R_{AB}}$$

Slika 29: Prikaz enačbe hamitoniana

V enačbi prikazujemo kinetično energijo elektronov v orbitali, kinetično energijo jeder, pozitivno privlačno silo negativnih elektronov in pozitivnih jeder, interakcijo elektronov in elektronov in interakcijo jeder. Naš cilj je dobiti časovno neodvisno Schrödingerjevo enačbo. Ta je HΨ=EΨ. Pri čemer je H že definiran v enačbi. H lahko predstavljamo kot matrico, zanima pa nas najmanjši H , da predstavimo najmanjšo eigen-vrednost E, da se enačba izide. Tako ugotavljamo hitrost reakcije, ki je pomembna. Iz teh informacij lahko vnaprej predvidevamo potek reakcije. To velja za eksponenten problem za klasične računalnike.

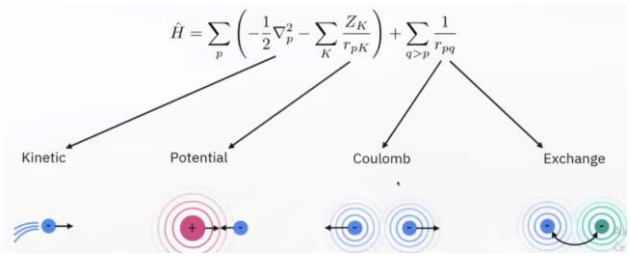
4.3.2 Elektronska struktura

Problema se lotimo:

- Z driverjem za prevajanje klasičnega programa v kvantni program.
- Skupnimi značilnostmi, ki jih poznamo že vnaprej.
- Na koncu potrebujemo še fermionske operatorje.

Prvi problem za prikaz simulacij kvantnega računalnika je elektronska struktura.

Za reševanje problema uporabimo driver, ki nam poda skupino lastnosti, kot so naboj, hitrost, itn. Nato s transformatorji lahko spreminjamo problem. Pošljemo rezultati v fermionske operatorje, ki so faza med problemom in fazo reševanja. Tako preusmerimo podatke v konverter, ki mapira problem na qubite in to pošljemo v operatorje, ki pošljejo problem v algoritem. Kadar nas zanima elektronska struktura, moramo vedeti eigen-vrednost hamiltoniana našega sistema.



Slika 29: Prikaz hamiltoniana molekule

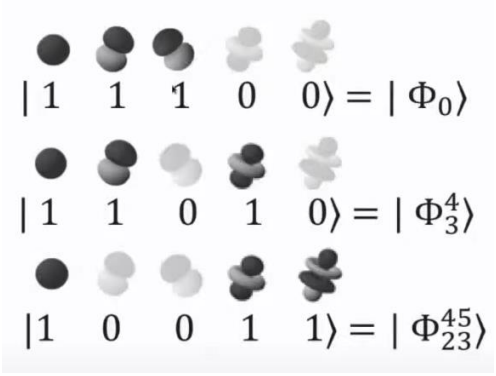
Hamiltonian je zgrajen iz kinetične energije elektronov, potencial, ki predstavlja delovanje jedra na elektrone, na koncu pa še delovanje med različnimi elektroni (kulon in menjava).

Kar naredimo z Schrödingerjevo enačbo:

$$\hat{H}|\Psi_n\rangle = E_n|\Psi_n\rangle \; E_0 = \frac{\langle\Psi_0|\hat{H}|\Psi_0\rangle}{\langle\Psi_0|\Psi_0\rangle}$$

Najdemo najnižjo energijo najbolj stabilnega sistema.

Orbitale v sistemu lahko predstavljamo z zapisom:



Kjer je stanje 0 najnižje energetsko stanje (ground state), saj elektroni zasedajo najmanjše orbitale. Nižji stanji sta v vzhičenem stanju, kjer je energija večja.

V naš program lahko dodamo še transformatorje, ki razdelijo ovojnico na zasedene in proste orbitale. Ker sklepamo, da med obema vrstama ne bo interakcij, prepustimo del zasedenih orbital klasičnemu računalniku in del prostih orbital kvantnemu računalniku. Tako ima kvantni manj dela s simulacijami, ki so enostavne. Temu rečemo aktiven prostor.

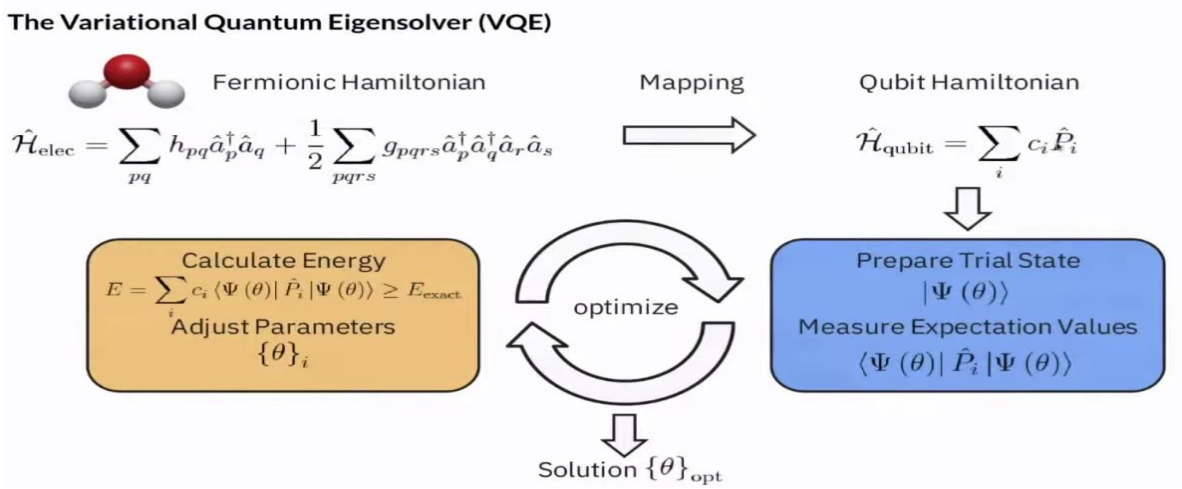
4.3.2.1 Mapiranje v prostor qubitov

Kot vemo iz kemije, ima vsak atom določeno elektronsko razporeditev. Za simulacije nas tako zanimajo molekulske orbitale. Te ločimo v orbitale (1S, 2p, 3d, …). Problem je, kako prenesti te informacije o elektronski postavitvi v kvantnih prostor, da jih kvantni računalnik lahko razume. Za to uporabljamo atomske orbitale, vsaka pa je sestavljena iz določenih funkcij, ki jih linearno združimo. Te funkcije so izbrane v osnovne sete po izbiri. Mi bomo uporabljali sto3g, ki je zelo groba ocena, ampak za naše primere dovolj enostavna.

Za nadalje mapiranje uporabljamo dejstvo, da je vsaka orbitala sestavljena iz elektronov v dveh spinih (gor ali dol). Temu rečemo spin-orbitala, iz njih pa je sestavljena polna molekularna orbitala, za mapiranje na kvantni računalnik. Poglejmo enostaven primer Jordan-Wigner mapiranja, kjer predstavimo zasedenost vsake spin orbitale z enim qubitom, ki ga predstavlja spin qubita. Poznamo še druge, bolj učinkovite metode mapiranja, kot je parity mapiranje. Ta za našo simulacijo vode porabi le 4 qubite, namesto 6 pri JW mapiranju.

4.3.2.2 Rešitev

Sedaj moramo poiskati rešitev našega problema, torej osnovno stanje (ground state). Za to uporabimo kvantni algoritem. Uporabimo že omenjeni variacijski algoritem. Iščemo najmanjšo eigen-vrednost, ki je na določeni valovni dolžini najmanjša od vseh. Torej variacijski algoritem išče najbližjo vrednost, glede na valovno dolžino in najde najmanjšo energijo.



Slika 30: Prikaz cikla delovanja algoritma za iskanje najnižje energijske vrednosti

Večji del smo že opravili, sedaj se še lotimo VQE algoritma. Zdaj iz hamiltoniana qubitov vzamemo Paulijevo vrednost, ki jo vstavimo v enačbo s poskusnim stanjem Ψ, ki je parametriziran z vrednostjo θ. Tako izmerimo pričakovano vrednost in podatke dodamo v klasičen računalnik. Ta izračuna energijo s pomočjo Hamiltoniana. Tako prilagodimo vrednost θ, ki jo vrnemo v kvantni računalnik. To delamo v loopu, dokler ne dobimo optimalnih parametrov θ, ko iz njih izračunamo energijo.

V namen raziskave bomo simulirali molekulo vode na simulatorju za kvantni računalnik. Naš program je dokaj preprost, saj uporabljamo vnaprej narejene driverje in solverje. Celoten

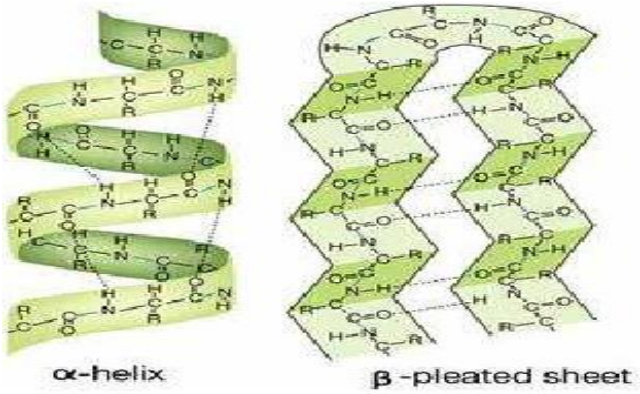
program in potrjeni rezultati so v **prilogi 7**. Eksperiment smo poskušali še z drugimi spojinami, vendar smo vseeno imeli omejitve pri izbiri kompleksnejših molekul, zato je bila voda spojina z največ orbitalami v naših eksperimentih. Kot vidimo, je naš program uspešno našel najnižjo energijsko vrednost, ki se sklada z vnaprej znanimi podatki.

4.3.3 Prelaganje beljakovin

Naše telo je pretežno sestavljeno iz beljakovin. Telo jih proizvaja ves čas in za delovanje potrebujemo specifične vrste. Obstaja le 20 različnih aminokislin, ta pa se povežejo v daljše verige s peptidnimi vezmi. Beljakovine so narejene iz ene ali več polipeptidnih verig. Poleg zgolj konfiguracije polipeptidov (torej zaporedja aminokislin), upoštevamo tudi obliko. Oblika je pomembna za delovanje na več različnih ravneh. Na primer encimi ne morejo pospeševati reakcij, če se oblika ne prilega. Encimi so pogosto beljakovine.

Poznamo 4 različna stanja ali oblike beljakovin: primarno, sekundarno, terciarno in kvartarno.

V primarni strukturi so različne aminokisline vezane v verigo, to je preprosta oblika. V sekundarni strukturi se verige prelagajo, najpogosteje v alfa-heliks in beta-nagubano obliko. Do tega pride predvsem zaradi vodikovih vezi različnih aminokislin.



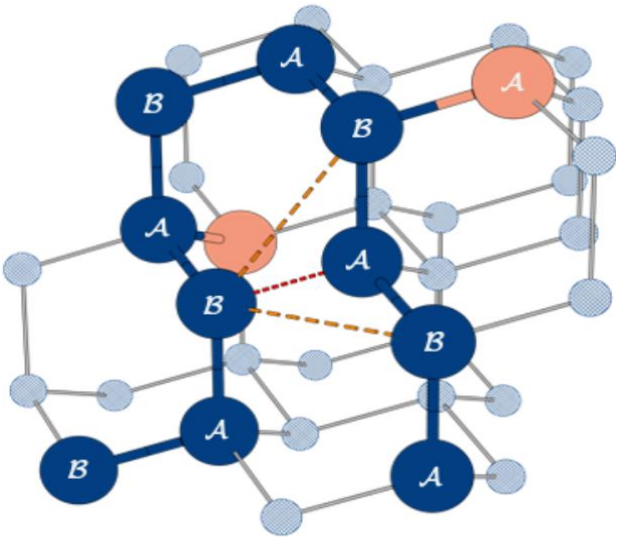
Slika 31: Prikaz helix in nagubane oblike beljakovin

V terciarni strukturi pride do 3D prelaganja verig sekundarne skupine zaradi R skupine ali stranske verige vsake aminokisline. Tu vplivajo faktorji kot so hidrofobnost in hidrofilnost R skupine, ionske vezi, Van der Waalsove interakcije, vodikove vezi, itn. V kvartarni obliki se več verig terciarne oblike združi v en funkcionalni protein.

Danes znanstveniki iščejo načine pravilnega prepogibanja beljakovin za želene rezultate

delovanja beljakovin. S simulacijami bi lahko lažje, ceneje in hitreje našli pogoje in oblike beljakovin za delovanje beljakovine. Z boljšim razumevanjem prelaganja beljakovin bi naredili napredek na razvoju novih cepiv, zdravljenju bolezni kot je Alzheimerjeva bolezen, na področju gojenja pridelkov, itn. Preloženi proteini imajo veliko različnih konfiguracij polipeptidnih vezi. Za 100 aminokislin veliko beljakovino obstaja 10^{47} conformacij. Beljakovina se lahko prepogne v njeno običajno naravno obliko v nekaj sekundah. Z večanjem števila aminokislin in večanjem medatomskih interakcij se problem za klasičen računalnik eksponentno veča, na kvantnem pa se težavnost veča linearno.

Naš cilj je najti najmanjšo energijsko konformacijo, tako da začenjamo z naključno in optimiziramo konformacijo glede na njeno energijo. Zdaj moramo problem kodirati v kvantni računalnik. To naredimo tako, da razdelimo delo na qubite, ki opisujejo konfiguracijo in relativno pozicijo posameznih aminokislin, interaktivni qubiti pa opisujejo interakcije med različnimi aminokislinami.



Slika 32: Prikaz medsebojnih povezav posameznih aminokislin v skupno beljakovino

Za naš poskus bomo vzeli verigo dolgo 7 aminokislin, te poimenujemo vsako s svojo črko. Poimenovanje je bolj kompleksno, zato ne bo razloženo v tej nalogi, vendar jih lahko poimenujemo s črkami APRLRFY. Okoli glavne verige aminokislin so lahko pritrjene tudi stranske verige. V našem primeru bodo te dolge le 1 aminokislino. Sedaj moramo računati interakcije med različnimi aminokislinami. To naredimo s predhodnim znanjem, izpeljanim iz quasi-chemical ocene. Te ocene sta prvič predstavila Miyazawa in Jernigan. Za splošno strukturo proteinov še lahko dodamo naključne vzorce interakcij.

Nato moramo še poskrbeti, da so zakoni fizike upoštevani, zato predstavimo kazensko funkcijo. Eno lahko predstavimo za pravilno kiralnost, druga preprečuje, da gremo dvakrat po isti verigi, kar prepreči, da bi se v simulaciji molekula prepognila sama vase in tretjo kazen za prekrivanje sosednjih aminokislin. Sedaj določimo še strukturo peptidov in definiramo problem z operatorji. Za reševanje raziskovalnega problema (najnižja energija, glede na zložen protein) uporabimo VQE algoritem s CVaR pričakovanimi vrednostmi za rešitev. Za klasično optimizacijo uporabljamo COBYLA. Namesto CVaR bi lahko uporabljali tudi bolj popularen QAOA, vendar je ta manj primeren.

Na koncu prikažemo graf naše simulacije. Z večanjem poskusov VQE algoritma, se najnižja energija približuje optimalni, tako kot je to predvideno v grafu. Program in graf sta priložena v **prilogi 8** raziskovalne naloge.

4.3.4 Simulacija kemijske reakcije

V tem problemu simuliramo potek kemijske reakcije med litijevim in vodikovim atomom v molekulo litijevega vodika. Predstavljajmo si, da želimo simulirati kemijsko reakcijo. Ključno pri reakcijah je spreminjanje energijskih stanj. Reakcije so lahko endotermne ali eksotermne. To pomeni, da moramo izračunati začetno stanje energij in glede na obnašanje energije ugotoviti, kako je potekala reakcija. Z vsakim novim delcem se računska zahtevnost eksponentno poveča. Naš algoritem deluje na osnovi variacijske metode. To pomeni, da s kvantnim računalnikom določimo energijsko stanje sistema s sklepanjem glede na valovno dolžino svetlobnih valov. Z VQE algoritmom določamo energijo sistema. Valovno dolžino svetlobe spreminjamo, dokler ne pridemo do minimalnega Hamiltoniana (matematični izraz za celotno energijo sistema - Hamiltonian lahko predstavlja energijo molekule, žoge na vzmeti, itn.). Gre za hibriden algoritem, kar pomeni, da delno deluje na klasičnem računalniku. Kvantni računalnik izračunava energijo, klasičen del pa optimizira spremenljive parametre. Primer uporabe je izračunavanje energije, glede na medatomsko dolžino v molekuli, kar bomo zdaj tudi pokazali. Najprej potrebujemo sklep o valovni dolžini. Na ta sklep bodo vplivali geometrija molekule, elektronska ovojnica in število elektronov. Ansatz je izraz za sklep o svetlobnih valovih glede na parametre molekule. To moramo še kodirati v kvantni računalnik, temu procesu rečemo mapping. Nato, glede na Ansatz, kvantni računalnik naredi merjenja in izračun ter pošlje te informacije v klasični računalnik, ki prilagodi Ansatz glede na nove podatke. Ko

najdemo minimalno energijsko vrednost na tej razdalji, se računalnik premakne na naslednjo razdaljo in tam izračuna energijsko vrednost. Na koncu simulacije vidimo, da je iz začetnega ansatza algoritem pravilno ugotovil dejansko energijo sistema, ki nam je bila vnaprej podana.

Podrobnejši postopek in program je v **prilogi 9** raziskovalne naloge.

V vseh simulacijah, ki smo jih izvedli, je kvantna simulacija pričakovano dobila iskan rezultat, glede na dane podatke. Za primerjavo s klasičnim računalnikom bi lahko uporabljali tehniko simuliranja (HOLOBAR, 2016) kot je Monte Carlo tehnika ali katero drugo s področja molekularne dinamike.

4.4. Kvantno strojno učenje[4]

Kako naučiti računalnik, da se nauči poiskati vzorce v danih podatkih in nato na osnovi ugotovljenega naredi sklepe. Lahko si jo zamislimo kot približevanje in optimizacijo funkcije. Torej lahko sklepamo, da v naravi obstajajo funkcije oziroma seti pravil, ki nam omogočajo interpretacijo vhodnih podatkov. Računalniki se morajo ta pravila za interpretacijo naučiti iz informacij in podatkov v razumljivi smeri, tako da so sposobni priti do zaključkov. Ljudje smo naravno sposobni prihajati do zaključkov iz vhodnih podatkov, ker smo skozi evolucijo naravno programirani s pravili in funkcijami, čeprav sami ne vemo, kaj ta pravila so. Predstavljajmo si funkcijo s pravili g, kjer so x vhodni podatki. g(x). S strojnim učenjem poskušamo narediti funkcijo f čim bolj podobno funkciji g, tako da vnesemo pod množico vseh podatkov x (nikoli nimamo na voljo vseh možnih podatkov za določen x). Najpomembneje pa je, da je funkcija f parametrizirana. f($\underline{x}$,Θ). Parametre lahko sproti reguliramo in optimiziramo, tako da dobimo najboljši približek funkciji g. Naš cilj pri strojnem učenju je torej najti optimalne parametre za funkcijo f, da bo čim bolj podobna naravnim pravilom ti. funkcije g. Druga naloga oz. problem pri ustvarjanju strojnega učenja je izbira modela funkcije f. Izbira modela je odvisna od več kriterijev, podrobneje v naslednjem poglavju.

Za strojno učenje potrebujemo:

1. **Podatke:** za iskanje vzorcev potrebujemo čim več podatkov, ki so lahko v obliki videov, slik, preglednic, grafov, itn. Izbira vrste podatkov je odvisna predvsem od modela. Kako računalnik bere podatke kot je slika? Piksli slike so predstavljene kot številke (po navadi

[4] Vir za to poglavje je bil Quantum Machine Learning - naveden kot vir 4.

0-255, saj so 8-bitne, čeprav enako velja za katero koli prikazovanje natančnost barv). Tako lahko sliko kot skupek barvnih pikslov predstavljamo kot numerične vrednosti. Sliko lahko predstavljamo kot vektor, v vektorski notaciji. Značilnosti (feature) podatkov pa nam povedo, koliko vrednosti je v neki podatkovni točki. Včasih značilnosti pravimo tudi dimenzija podatkovne točke. N število vektorjev predstavlja podatkovni nabor. Podatkovni nabor lahko predstavljamo tudi kot matrice, kjer vsak stolpec pomeni vsako podatkovno točko.

2. **Model:** Model vzame vnešene podatke x in jih s funkcijo in dodatnimi parametri spremeni v neko napoved, oznako, output, distribucijo oz. y.
3. **Stroškovna funkcija** (cost function): Ta funkcija primerja naš rezultat (y) s pravilnim rezultatom in določi natančnost naše f funkcije. Zamislimo si jo kot C(f(x , Θ) pravilni rezultat). Tako dobimo izid, kako natančen je naš trenutni model. Za to si lahko predstavljamo formulo ($\hat{y}$ -y) $(\hat{y} - y)^2$. Višji izid, ki ga model dobi, slabše opravlja.

Strojno učenje uporabljamo v vsakodnevnem življenju. Primeri: spam email, vreme, clustering, prepoznavanje pisave, ustvarjanje novih podatkov na osnovi učenja starih (obrazi).

V strojnem učenju lahko določamo 3 sisteme, načine učenja za prepoznavo podatkov.

1. **Nadzorovano učenje:** podatke označimo z oznakami splošnih značilnosti v data pointu. V nadzorovanem označimo vse vrste podatkov in glede na pravilnost sklepa modela. Tako naredimo podatkovne informacije o opredelitvi modela strojnega učenja. Nato model popravljamo glede na pravilnost.

   Torej s pravilno izbrano Θ je y bliže pravilni vrednosti, glede na skupno povprečje pravilnosti klasifikacije. Θ mora biti linearni vektor, velikosti odvisne od podatkovne točke v data setu. Pri procesu optimizacije moramo izbrati Θ. Kako to naredimo? Predstavljajmo si funkcijo, ki prikazuje cost modela v 2 dimenzionalnem grafu, kjer je na x osi Θ. Kjer je ta funkcija y minimalna, tam je x Θ. Kako to določiti, če takšnega grafa nimamo. Prav nam pride Gradient-descent algoritem. Začnemo z izbero naključne vrednosti Θ. Nato izračunamo Derivator cost funkcije / Derivator Θ. To nam poda gradientni vektor. Algoritem nato nadgradi parametre v nasprotni smeri od gradienta. Problem pri tem algoritmu nastane, če se zataknemo na lokalnem minimumu, ki pa ni enak globalnemu. V resnici so grafi cost funkcije veliko bolj kompleksni v več dimenzijskem prostoru. Za to uporabimo Stochastic gradient-descent.

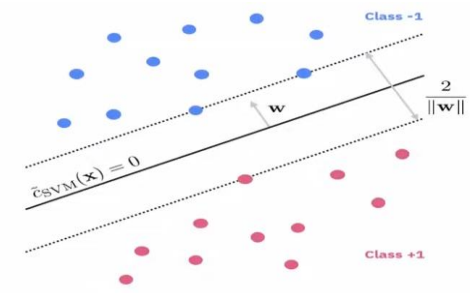
2. **Nenadzorovano učenje:** v nenadzorovanem učenju v model damo podatke, ki nimajo neke oznake z vidika človeka, ki je te podatke dal v model. Nato model najde skupne značilnosti podatkov, ki nam niso tako očitne in glede na značilnosti podatke opredeli.
3. **Utrjevalno učenje:** utrjevalno učenje vsebuje agenta, ki v svetu, kjer so določena pravila, skuša doseči najboljšo možno nagrado in temu prilagaja aktivnosti v tem okolju.

Poznamo več različnih vrst modelov za strojno učenje. Najboljšega izberemo s pomočjo generalizacije. Vprašamo se, kako dober bo model na še ne videnem podatkovnem setu. Pri tem je pomembno najti pravo ravnotežje med razhajanjem (variance) in pristranskostjo (bias). Če je model prekompleksen, ima previsok variance, bo vsak podatek, ki ima majhno odstopanje, spregledan. V nasprotnem primeru, s preveliko nagnjenostjo (biasom), pa model ignorira naše podatke pri treningu.

4.4.1 Kernel strojno učenje

4.4.1.1 Klasifikacija

Za vpogled v kvantno strojno učenje si vzemimo osnoven primer nadzorovanega binarnega klasificiranja. To pomeni, da okarakteriziramo podatkovne točke v neki podatkovni bazi glede na prej naučene značilnosti. Predstavljajmo si bazo podatkov z vektorjem, kateremu lahko pripišemo 2 lastnosti. Vektorje lahko predstavimo na dvodimenzionalnem prostoru, kjer točke predstavljajo dve lastnosti. Cilj našega modela je, da najde najboljšo ločnico med dvema skupinama točk. Za to najprej potrebujemo model za trening, kjer naš model išče ločnico, in za model za testiranje, kjer lahko preverimo natančnost ugotavljanja modela s preverljivostjo podatkov. V setu treniranja je vsak vektor (točka na grafu) označena s +1 ali -1. V testnem setu oznaka modelu ni dana in mora razvrstiti vektorje sam. Ločnico lahko določimo na več načinov, eden je z algoritmom Support Vector Machine, ki najde najbližja vektorja in glede na njiju.



Slika 33: Prikaz binarne klasifikacije s algoritmom SVM

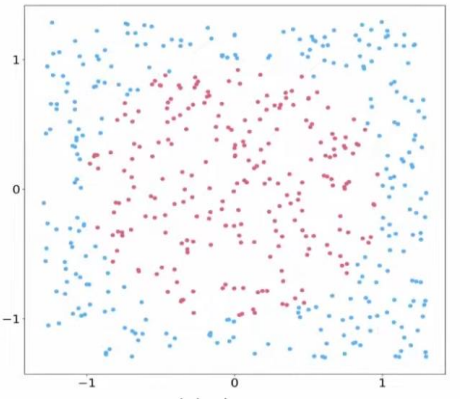
Torej, želimo, da je naša ločnica maksimalno oddaljena od najbližjih točk (vektorjev) na grafu. Matematična definicija problema:

$$\tilde{c}_{\mathrm{SVM}}(\mathbf{x}) = \mathrm{sign}(\mathbf{w}^T\mathbf{x} - b)$$

$$\min_{\mathbf{w}\in\mathbb{R}^s,\, b\in\mathbb{R}} \|\mathbf{w}\|^2$$

$$\text{s. t. } y_i(\mathbf{w}^T\mathbf{x}_i - b) \geq 1$$

Problem nastane, če je risanje črte med različnimi skupinami vektorjev nemogoče, torej dveh različnih klasov ne loči jasna ločnica.



Slika 34: Prikaz morebitne baze podatkov na koordinatnem sistemu

V tem primeru moramo ločnico potegniti v višji, nelinearni, dimenziji. To naredimo tako, da celotno bazo podatkov transformiramo v višjo dimenzijo. Tako spremenimo prvotno enačbo v:

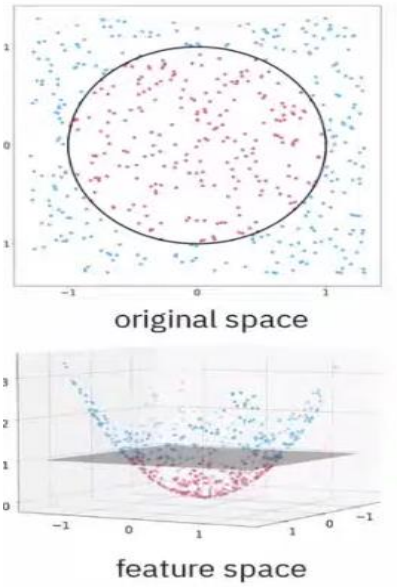
$$\tilde{c}_{\mathrm{SVM}}(\mathbf{x}) = \mathrm{sign}(\langle \mathbf{w}, \phi(\mathbf{x})\rangle_v - b)$$

S kernelom lahko preuredimo problem tako, da velja zgolj za vrednosti kernela. Zato ni potrebe po računanju vsakega vektorja posebej v nov dimenzionalni prostor.

Poglejmo primer:

Našo prvotno mapo lastnosti smo s kernelom $k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}') = \mathbf{x} \cdot \mathbf{x}' + \|\mathbf{x}\|^2\|\mathbf{x}'\|^2$ preuredili tako, da imamo 3 lastnosti o vsaki točki: $\phi(\mathbf{x}) = (x_1, x_2, x_1^2 + x_2^2) \in \mathbb{R}^3$.

Tako lahko prikažemo našo mapo:



Slika 35: Prikaz prehoda v višjo dimenzijo za klasifikacijo podatkov s hiper-ravnino

Zdaj lahko določimo ravnino oziroma hiper-ravnino (če govorimo o večdimenzionalnem prostoru). Celoten proces optimizacije poteka na klasičnih računalnikih, saj so kvantni učinkoviti le za diskretne optimizacijske naloge, to pa je primer neprekinjene optimizacije (continuous). Zato velja kvantno strojno učenje za hibridni algoritem, saj je en del problema rešen na klasičnih in en del na kvantnih računalnikih.

Kvantni računalniki pridejo v uporabo, ko računamo kernel za prevod vektorjev v višjo dimenzijo. Enačba za prevod mape iz klasičnega zapisa v višjo dimenzijo:

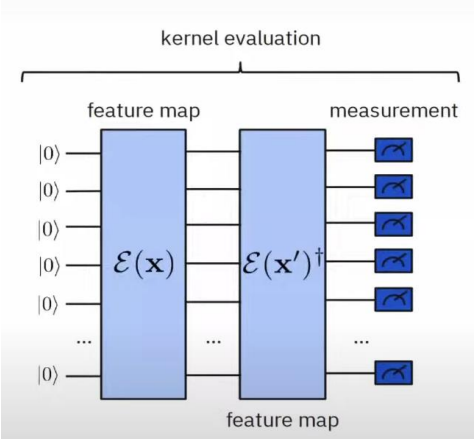
$$\begin{aligned} \psi : \mathbb{R}^s &\rightarrow \mathcal{S}(2^q) \\ \mathbf{x} &\mapsto |\psi(\mathbf{x})\rangle\langle\psi(\mathbf{x})| \end{aligned}$$

Tako za izračun kvantnih kernelov dodamo obe mapi v enačbo:

$$\begin{aligned} k(\mathbf{x}, \mathbf{x}') &= \mathrm{tr}[|\psi(\mathbf{x}')\rangle\langle\psi(\mathbf{x}')| \, |\psi(\mathbf{x})\rangle\langle\psi(\mathbf{x})|] \\ &= |\langle\psi(\mathbf{x}')|\psi(\mathbf{x})\rangle|^2 \\ &= |\langle 0|\mathcal{E}(\mathbf{x}')^\dagger \mathcal{E}(\mathbf{x})|0\rangle|^2 \end{aligned}$$

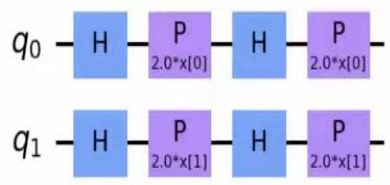
Ali poenostavimo formulo: $k(\vec{x}_i, \vec{x}_j) = \langle f(\vec{x}_i), f(\vec{x}_j) \rangle$, kjer je k kernel funkcija, x_i in x_j inputa iz originalne dimenzije in f mapa iz n v m dimenzijo. Kernel funkcijo lahko predstavljamo tudi kot matrico.

Uporabljamo mape:



Slika 36: Nazoren prikaz kvantnega kroga s prehodi skozi dano mapo lastnosti

Vzemimo primer klasifikacije. Vzeli bomo ad hoc podatkovni set, ki je na voljo na linku. V pomoč nam bo še scikit-learn knjižnica za SVC algoritem. Izbrali bomo zFeature mapo, saj je izbira mape, ki nalaga podatke v kvantni računalnik ključna.



Slika 37: Primer zFeature mape, ki jo uporabljamo.

Za učenje modela uporabljamo SCV algoritem. Kernel lahko vstavimo kot klicno funkcijo ali kot vnaprej izračunano matrico. V našem primeru jo bomo vstavili kot klicno funkcijo.

Sedaj lahko uporabimo še kvantni algoritem QSVC.

Eksperimente smo delali tako, da smo primerjali reševanje klasičnega SVC in kvantnega QSVC algoritma. Tokrat so nas rezultati ponovno presenetili. QSVC algoritem na simuliranem kvantnem računalniku je za več različnih testiranj podatkov potreboval enako časa kot SVC algoritem na klasičnem. Sicer drži, da je bila izvedba programa prilagojena za kvantni računalnik in da obstajajo boljši načini reševanja problemov za klasični računalnik, vendar

rezultat, ki je bil za navrh še 100% natančen, kaže na sposobnost kvantnih algoritmov. Natančnost je ostala enaka, ne glede na to, na kolikšni bazi je bil model treniran, saj je od 20 treniranih primerkov bil model 100% natančen.

4.4.1.2 Združevanje

V tem primeru podatke, glede na njihove lastnosti, karakteriziramo v skupine podatkov na mapi podatkov. Računalnik določi meje skupin podatkov, glede na predhodno učenje in tako karakterizira nove podatke. To lahko določimo z matrico za združevanje podatkov. Za oceno natančnosti ne potrebujemo testne skupine, temveč lahko podatke preverimo na dani osnovi.

4.4.2 Klasifikacija in regresija

V nadziranem strojnem učenju za analiziranje podatkov uporabljamo 2 načina analize:

1. V klasifikaciji podatke razvrstimo v različne klase (razrede), ki imajo svojo lastnost. Primer klasifikacije je razvrščanje slik v skupine (razrede) mačk in psov. Model glede na značilnosti novih podatkov in treninga na predhodnih podatkih določi razred in da oznako vsaki podatkovni točki, pri tem pa še lahko zapiše verjetnost.
2. V regresiji algoritem išče konkretno verjetno vrednost druge lastnosti podatka, glede na učenje polnih vrednosti predhodnih podatkov. Podatke zapisujemo kot vektorje različnih vrednosti.

4.4.2.1 Vrste modelov

Linearni model

Išče linearno funkcijo kot premico (2 dimenziji) ali kot hiper-ravnino (več dimenzij), glede na vrednosti podatkov. Nato za novo vrednost podatka najde drugo vrednost glede na znano funkcijo. Lahko zapišemo kot $y=\theta x$. V problemu klasifikacije pa funkcija deli podatke v klase.

Nevronske mreže

So sestavljene iz več vozlišč, ki so razporejene v več plasti. V vsaki plasti vozlišča prilagodijo parametre funkcije, tako da najdejo lastnosti podatka, ki ustrezajo lastnostim med fazo treniranje.

4.4.3 Primerjava strojnega učenja na kvantnem in klasičnem računalniku

Ker so kvantni računalniki v zgodnjem razvoju, bomo za primerjavo dela na klasičnem računalniku uporabili grafično kartico za širši potrošniški trg, medtem ko bomo za kvantni računalnik uporabili IBM-ov qasm simulator, simuliran lokalno na procesorju. Pogledali bomo razliko v treniranju nevronske mreže na podatkovnem setu MINST, ki shranjuje ročne zapise. Za ohranjanje preprostosti bomo uporabili samo števki 0 in 1, čeprav baza vsebuje podatke za druge zapise. Treniranje bomo izvedli na 100 primerih in testiranje na 50. V obeh primerih uporabljamo integracijo z okvirjem PyTorch, s tem, da v kvantnih nevronskih mrežah uporabljamo hibridne nevronske mreže iz SDK qiskit, ki so integrirane v PyTorch modul za učenje. Kot vidimo, je treniranje za klasični računalnik vzelo 2,5 sekundi, z večanjem baze podatkov pa se čas linearno veča. Preizkusili smo še 1000 in 5000 podatkov, testiranje pa je v obeh primerih ostalo na 150 primerih. Pri tem smo ugotovili minimalno odstopanje natančnosti (99,9% in 99,9%), vendar je to bilo pričakovano zaradi večje baze podatkov za testiranje. Zaključimo lahko, da je 100 primerov pri treniranju dovolj za zelo natančen model in da čas treniranja ne odtehta natančnosti, ki je ostala skoraj enaka tudi pri 150 primerih testiranja. Pri kvantnem strojnem učenju vidimo, da je natančnost modela prav tako 100%, kar je spodbudno. Model je za treniranje sicer potreboval 83 sekund, vendar je to bilo pričakovano, saj je kvantni računalnik bil simuliran na CPE. Model je moral skozi cikle treniranja iti večkrat, za minimiziranje izgube, kot to prikazuje graf in loss funkcija glede na delež treniranja. Zaključimo lahko, da so kvantne nevronske mreže že na ravni, ko v znosnem času lahko opravijo nalogo strojnega učenja z visoko natančnostjo. Kljub temu, da je algoritem, v našem primeru hibriden (optimizator je še vedno klasičen), lahko kvantne nevronske mreže pospešijo osnovne probleme prilagajanja parametrov z možnostjo superpozicije.

4.5 Kvantna kriptografija

Z vse večjo uporabo interneta in pošiljanja, z vse več sporočili (l. 2020 je uporabljalo internet 4.5 milijarde uporabnikov), je enkripcija (za vsa naša sporočila, bančna nakazila, itn.) vse pomembnejša. V osnovi deluje tako, da naprava prejme šifrirano sporočilo, ki izgleda povsem brez pomena, in ga preko ključa prevede v sporočilo, ki ga lahko preberemo. Enkripcija je zavarovana preko enosmerne funkcije, funkcije, ki jo je lahko izračunati, vendar težko razstaviti. Za generacijo ključev se uporablja množenje dveh praštevil. Ker so ključi danes zelo

kompleksni (več kot 1000 bitov, primer: 1024 bitov je 309 mestno število v decimalnem sistemu), kar je lahko problem tudi za najboljše računalnike na svetu. Za dešifriranje bi kvantni računalniki uporabljali Shorov algoritem. Vendarle smo lahko precej dolgo časa brez skrbi, da bi kdo lahko s kvantnim računalnikom prebral našo e-pošto, saj dandanes niso niti približno na dovolj visoki ravni, da bi bili sposobni prakticirati nekaj tako kompleksnega, kot Shorov algoritem.

Danes za enkripcijo uporabljamo RSA protokol, kjer je sporočilo zavarovano z zelo velikim pol-praštevilom (številom, ki ima 4 popolne delitelje). Sporočilo lahko dekodiramo, če imamo zaseben ključ, ki ga zmnožimo z javnim. Da računalnik uporablja sporočilo brez zasebnega ključa, mora najti prafaktorje števila, kar je zahteven proces. Trenutna najboljša rešitev je, da računalnik ugiba naključna števila in preizkuša, ali je število prafaktor danega števila. Ta proces je zelo počasen, tudi z največjimi superračunalniki. Današnja enkripcija podatkov ne zmore v celoti zavarovati pred dešifriranjem, vendar pa lahko to oteži tako, da za to potrebujemo veliko časa in računalniške moči. Kako velika števila se uporabljajo, mora biti ravnotežje med hitrostjo množenja velikih števil (ključev) in varnostjo.

4.5.1 Shorov algoritem

S Shorovim algoritmom in kvantnimi računalniki bi proces iskanja prafaktorjev bil veliko hitrejši. Prav tako kot klasični tudi Shorov kvantni algoritem začne z naključnim ugibanjem. Nato Shorov algoritem spremeni ugibanje v precej boljše ugibanje, ki je verjetno končna rešitev. Naše ugibanje je dobro že, če si število g deli nekatere prafaktorje. Nato uporabimo Evklidov algoritem, s katerim lahko pridemo hitro do rešitve. To je pri veliko številih še zmeraj malo verjetno. Zato uporabimo previlo, da če eno število množimo s seboj dovoljkrat, bomo dobili večkratnik drugega števila +1. To lahko zapišemo kot $(g^{p/2+1})(g^{p/2-1}) = mN$. Tako je 37,5% ugibanj dejansko prafaktor iskanega števila (izjeme, ko je p liho in g je večkratnik N). Problem je najti p (kolikokrat množimo g s samim sabo). Tu uporabimo kvantne računalnike, ki s superpozicijo ugibajo rezultat za več števil hkrati in z interferenco najdemo en rezultat z največjo verjetnostjo. Predstavljajmo si p kot frekvenco, ali pa pridemo do faktorjev praštevil na drug način.

Za iskanje prafaktorjev velikega števila N, ugibamo g število. Če ima g skupne prafaktorje z N, smo končali. Zelo verjetno pa je, da jih nima. V tem primeru izberemo p, ki je potenca na g.

Potenciran g si deli prafaktorje z N.

If $\mathbf{GCF}(g, N)! = 1$:

$g^p = m^* N + 1$
$g^p - 1 = m^* N$
$\left(g^{p/2} + 1\right)\left(g^{p/2} - 1\right) = m^* N \, \mathbf{GCF}\left(g^{\mathrm{p}/2} \pm 1, N\right) = \text{ factors of } N$

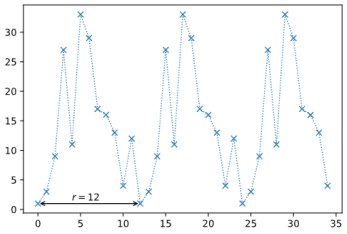
Kako pa ugotovimo p?

S Shorovim algoritmom iščemo p, da bo $g^p - 1 = \left(g^{p/2} + 1\right)\left(g^{p/2} - 1\right) = m^* N$. P ne sme biti prevelik, da ne dobimo kot rezultat večkratnikov N in ne sme biti liho, saj bi nastali ulomki.

Problem si predstavimo drugače: $g^p (\mathrm{mod}\ N) = 1$mod nam pove, koliko je ostankov po deljenju.

Shorov algoritem izkorišča dejstvo, da se pri (mod n) vsakih n pojavlja isto zaporedje.

Primer:



Tako Shorov algoritem spremeni problem v iskanje zaporedij.

$$g^p (\mathrm{mod}\ N), g^{2^* p} (\mathrm{mod}\ N), g^{3^* p} (\mathrm{mod}\ N), \ldots, g^{k^* p} (\mathrm{mod}\ N) = 1$$

Sestavimo Shorov algoritem:

1. S H operacijo spremenimo qubite v superpozicijo za kandidate za p. Izračunamo vrednosti za vsak g^p (mod N).
2. Najdemo interval, kjer je rešitev 1. Kvantni računalniki s QFT hitro najdejo funkcijsko periodo. QFT je pomemben del in je njegova sestava definirana.

   $\left(g^{p/2} + 1\right)\left(g^{p/2} - 1\right) = m^* N \rightarrow$
   $\mathbf{GCF}\left(g^{p/2} \pm 1, N\right) = \text{ factor of } N$
   $\mathbf{GCF}\left(7^{4/2} \pm 1, 15\right)$
   $\mathbf{GCF}(49 + 1, 15)$
   $\mathbf{GCF}(50, 15) = 5$

Zato določimo unitarni operator za ocenjevanje faze. $U|y\rangle \equiv |ay \bmod N\rangle$. Superpozicija te funkcije

$$|u_0\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} |a^k \bmod N\rangle$$

pa bi bilo eigen-stanje.

Preizkusimo še delovanja našega programa za implementacijo Shorovega algoritma. Kot vidimo na primeru v **prilogi 16**, smo algoritem sestavili sami in ga testirali za iskanje prafaktorjev 15. Ker je za samostojno sestavljanje algoritma za vsako iskanje posebej zamudno, smo v drugih eksperimentih in meritvah uporabljali modul Shorovega algoritma iz knjižnice qiskit. Na začetku smo testirali še iskanje prafaktorjev s klasičnim računalnikom, vendar je pri znosno nizkih številih (pod 1 000 000) ostalo iskanje dovolj hitro, da bi bili naši rezultat neprimerljivi z rezultati iskanja na simulaciji Shorovega algoritma. Ta je pri nižjih številih (do 15) delal relativno hitro in natančno. Zataknilo se je pri večjih številkah, ko je za iskanje prafaktorjev števila 35 potreboval 22 minut. Poizkusili smo narediti še eksperiment na višjih številih, vendar je trajalo tako dolgo, da po 45 minut še ni bilo rezultatov. Zato smo se odločili, da bo število 35 najvišje število, ki bo zajeto v pregled.

5. REZULTATI IN RAZPRAVA

V nalogi so bili zaradi boljše preglednosti in števila obravnavanih problemov rezultati podani ob opisu problemov in v prilogah poleg programov. Kljub temu lahko v zaključku opravimo sintezo naših dognanj. V nekaterih problemih, ki smo si jih zastavili, so kvantni računalniki, glede na njihova pričakovanja in njihovo poznavanje v svetu računalniške znanosti, relativno razočarali. S tem predvsem mislimo na rezultate problema potujočega trgovca, kjer simulacija kvantnega računalnika v nekaj desetkratnem času ni zmogla najti najboljše poti. Raziskava tega problema nas je še posebej presenetila, saj smo te raziskave pričakovali največ od vseh. Verjamemo, da je navedena problematika kvantnih računalnikov posledica zgodnje faze v razvoju le-teh. Prepričani smo, da bo v nadaljevanju eksponentnega razvoja področja kvantnih računalnikov prišla do izraza veliko večja uporabnost in hitrosti, verjamemo, da na vseh področjih razvoja in aplikacij le-teh. Utemeljenost te teze nam kažejo tudi rezultati na drugih področjih raziskovanja v raziskovalni nalogi, kjer je kvantni računalnik veliko uspešnejši od klasičnega računalnika, kar kaže na svetlo prihodnost kvantnih računalnikov.

5.1 Povzetek rezultatov raziskave - analiza rezultatov raziskovanja po področjih

5.1.1 Optimizacija

	Simulacija kvantnega računalnika	**Rešitev na klasičnem računalniku**
Problem potujočega trgovca: Možnosti simulacije do 5 mest. Primerjava na 4.	Čas: 417s Cena: 302	Čas: 0,27s Cena: 236
Max-Cut problem	Čas: 2,45s Cena: 10,8	Čas: 0,2s Cena: 10,8
Problem preusmerjanja vozil	Cena: 23269	Cena: 127

5.1.2 Simulacije narave

	Reševanje simulacije kvantnega računalnika
Problem prelaganja beljakovin	Najdena rešitev je optimalna in se sklada z vnaprej znanimi podatki.
Elektronska ovojnica	Najdena rešitev je optimalna in se sklada z vnaprej znanimi podatki.
Simulacije kemijskih reakcij	Simulirana je bila kemijska reakcija med atomom litija in vodika. Pri tem smo iskali energijo sistema, s pomočjo katere predvidevamo potek celotne kemijske reakcije.

5.1.3 Strojno učenje

	Kvantna simulacija	**Klasično reševanje na GPE**
Kernel QSVM	Čas: 23,1s Natančnost: 100%	Čas: 23,2s Natančnost: 100%
Binarna klasifikacija	Natančnost: 80%	
Binarna klasifikacija pisave	Čas treniranja mreže: 83s Natančnost testiranja: 100%	Čas treniranja mreže: 2,5s Natančnost testiranja: 100%

5.1.4 Kriptografija

	Reševanje kvantne simulacije na qasm simulatorju
Shorov algoritem	Z večanjem števila, kateremu iščemo prafaktorje, je simulacija potrebovala veliko več časa. V primerjavi z iskanjem praštevil s klasičnim računalnikom, pa je bil algoritem vedno slabši. Takšni rezultati so pričakovani, saj ima kvantno računalništvo prednost pred klasičnim šele pri večjih številih. Problem takšne primerjave je, da s simulacijo ne moremo doseči dovolj velikih števil pri iskanju prafaktorjev, da bi sistem kvantnega reševanje bil boljši od klasičnega. Najvišje testiran število je bilo 35, zanj je simulacija potrebovala 22 minut.

6. ZAKLJUČEK

V nalogi smo pogledali veliko uporabnih primerov in zanje iskali rešitve tako s klasičnimi računalniki kot s kvantnimi računalniki. Razvoj povsem nove tehnologije kvantnih računalnikov je še v zelo zgodnji stopnji in za pričakovati je bilo, da bodo klasični v veliko primerih bolj konkurenčno reševali probleme. V raziskovalni nalogi smo ugotovili, da kvantno računalništvo veliko obeta in če se bo razvoj še naprej razvijal v eksponentnem tempu, bodo imeli kvantni računalniki zelo pozitiven vpliv na življenja vseh in to na vseh področjih družbe in razvoja. Uspeli smo dokazati, da imajo kvantni računalniki zelo velik potencial, vendar tak potencial prinaša tudi veliko odgovornost, predvsem s strani kibernetske varnosti in odgovornega ravnanja z možnimi posledicami, ki ga ta tehnologija prinaša.

Naše hipoteze in ugotovitve iz raziskave glede hipotez:

1. **Hipoteza 1:** Kvantni računalniki oziroma simulacije so sposobni reševati resnične probleme v konkurenčnem času.

   Da, kvantne simulacije, kljub temu da delujejo na klasični strojni opremi, lahko v konkurenčnem času natančno rešujejo probleme.

2. **Hipoteza 2:** Klasični računalniki imajo še zmeraj razvojno prednost, ki se zmanjšuje s hitrim razvojem kvantnih računalnikov.

   Da, pričakovano so klasični računalniki in algoritmi razviti zanje bili hitrejši od kvantnih računalnikov in simulacij kvantnih računalnikov. Prednost kvantnih bi lahko danes lažje iskali z večanjem problemov, vendar še nimamo dovolj sposobnih kvantnih računalnikov, ki bi delovali na dovolj velikem nivoju za prikaz prednosti pred klasičnimi.

3. **Hipoteza 3:** V kolikor algoritme kvantnih računalnikov simuliramo na klasičnih računalnikih, kvantni algoritmi ne morejo tekmovati z algoritmi, ki so prilagojeni samo klasičnim računalnikom - zato bodo klasične rešitve na klasičnih računalnikih zmeraj boljše od rešitev iz kvantnih simulacij na klasičnih računalnikih.

4. **Hipoteza 4:** Pri problemih optimizacije bo kvantna simulacija našla optimalno rešitev v precej daljšem času kot klasični računalnik, cena rešitve pa ne bo za veliko odstopala

od optimalne.

Ne, najdena rešitev v večini problemov ni bila optimalna, razlika med najdeno rešitvijo in optimalno pa se je večala s velikostjo primera.

5. **Hipoteza 5:** S stopnjevanjem problema optimizacije, se bo čas večal do limita simulacije, natančnost oziroma cena pa bo padala s večanjem problema.

   Da, čas se je z linearnim večanjem problema eksponentno večal. S simulacijo smo trčili v limit na prenizkem nivoju za prikaz prednosti kvantnih računalnikov pred klasičnimi.

6. **Hipoteza 6:** Pri simulacijah narave bodo rezultati simulacije kvantnega računalnika večinoma v prednosti.

   Da, vse simulacije s kvantnim računalnikom so v prednosti.

7. **Hipoteza 7:** V vseh primerih strojnega učenja bo kvantni računalnik počasnejši in zaradi hrupa manj natančen kot klasični. Razlika v natančnosti bo posebej velika pri večjih nevronskih mrežah.

   Ne, simulacije kvantnih računalnikov so sicer bile počasneje v primeru klasifikacije z nevronskimi mrežami, vendar sta v primeru SVM algoritma bila po času kvantni in klasični način povsem izenačena. Simulacije kvantnih računalnikov so bile enako natančne kot klasični, in sicer so bili 100%.

8. **Hipoteza 8:** Pri uporabi Shorovega algoritma bo kvantna simulacija natančna, zaradi potrebnega procesa v ozadju pa bo trajala precej dlje kot klasičen. Z večanjem števila se bo čas eksponentno večal.

   Da, simulacije kvantnih računalnikov so v vseh primerih pol-praštevil bile popolnoma natančne. Časovno je simulacija zaostajala predvsem zaradi neprilagojenosti klasičnega računalnika na kvantne algoritme in prenizkega števila za merjenje prednosti.

Naših eksperimentov na žalost nismo mogli izvajati na najnaprednejših kvantnih računalnikih, zato smo se morali znajti s manj sposobnimi, kakor tudi s simulatorji na klasičnih računalnikih. Pogoji eksperimentiranja so bili oteženi, kljub temu smo uspeli ugotoviti in raziskati veliko novega, še posebej, ker je to področje v zgodnji fazi razvoja. Ugotovili smo, da kvantni računalniki porajajo nove vrste problemov, katerih prej še nismo poznali. Dejstvo, da lahko kvantni hardware, ki je v razvoju šele zadnje desetletje, in v katerega je bilo vloženih komaj

nekaj milijard evrov, lahko konkurira klasičnemu, ki se razvija zadnjih 75 let in v katerega je bilo vloženih več bilijonov evrov, na raziskovanju in razvoju pa delalo več 100 tisoč ljudi, kar je neverjetno. Tovrstne raziskave spodbujajo razvoj področja, navkljub tveganjem, ki jih prinaša nova neraziskana tehnologija.

7. DRUŽBENA ODGOVORNOST

V raziskovalni nalogi raziskujem možnosti obdelave podatkov in izboljšanja procesov na različnih področjih družbenega delovanja z uporabo kvantnih računalnikov. V teoriji je podano predvidevanje, da naj bi z uporabo kvantnih računalnikov bistveno hitreje obdelali različne probleme, kar vpliva na družbeni razvoj celotne družbe in na vseh področjih delovanja. Tudi s svojo nalogo želim prispevati k razvoju znanja na tem področju. Raziskave s področja digitalizacije in kvantnih tehnologij prispevajo k trajnostnemu razvoju in širšemu družbenemu napredku, ključni in veliki pa bodo vplivi kvantnih računalnikov na trajnostni razvoj, ekonomijo, astronomijo, gospodarstvo, medicino, finance, naravo varstvo, meteorologijo, kmetijstvo, logistiko, farmacijo, robotiko, internet, programiranje, biologijo, matematiko, novi materiali, gnojila, kriptografijo, optimizacijo, itn. V Evropski uniji je kvantno računalništvo med prioritetami strategije razvoja EU, prav tako tudi v nekaterih drugih razvitih državah.

9. VIRI IN LITERATURA

1. HOLOBAR, Aleš, 2016, Kvantno računalništvo in kriptografija [na spletu]. Fakulteta za elektrotehniko, računalništvo in informatiko, Inštitut za računalništvo. [Dostopano 17 marec 2022]. ISBN 978-961-248-516-0. Pridobljeno s: https://dk.um.si/IzpisGradiva.php?lang=slv&id=61634
2. 2020, Learn Quantum Computation Using Qiskit [na spletu]. [Dostopano 17 marec 2022]. Pridobljeno s: https://qiskit.org/textbook/preface.html
3. Britannica, T. Editors of Encyclopaedia (2021, May 28). Schrödinger equation. Encyclopedia Britannica. https://www.britannica.com/science/Schrodinger-equation
4. Biamonte, J., Wittek, P., Pancotti, N. et al. Quantum machine learning. Nature 549, 195–202 (2017). https://doi.org/10.1038/nature23474
5. 2021, Qiskit: An Open-source Framework for Quantum Computing [na spletu]. [Dostopano 17 marec 2022]. Pridobljeno s: https://qiskit.org/documentation/

Priloga 1

Bernstein Vazirani

March 5, 2022

V oracle algoritmu Bernstein-Vazeranijev algoritem s pomočjo odkrivanja oracle poda, ugotavlja skrivno število. Sestava kroga je vedno enaka. Krog ima toliko qubitov kolikor je mest v številu +1, na zadnjega dodamo not vrata, nato pa na vse. Nato dodamo še controled-not vrata,katerih postavitev je odvisna od števila, ki iščemo. Da dobimo cele vrednosti na vse razen zadnjega qubita dodamo H vrata in krog izmerimo. Malo podrobneje o tem sicer preprostim algoritmom piše v nalogi.

[29]:
```
from qiskit import *
```

[30]:
```
%matplotlib inline
from qiskit.tools.visualization import plot_histogram
```

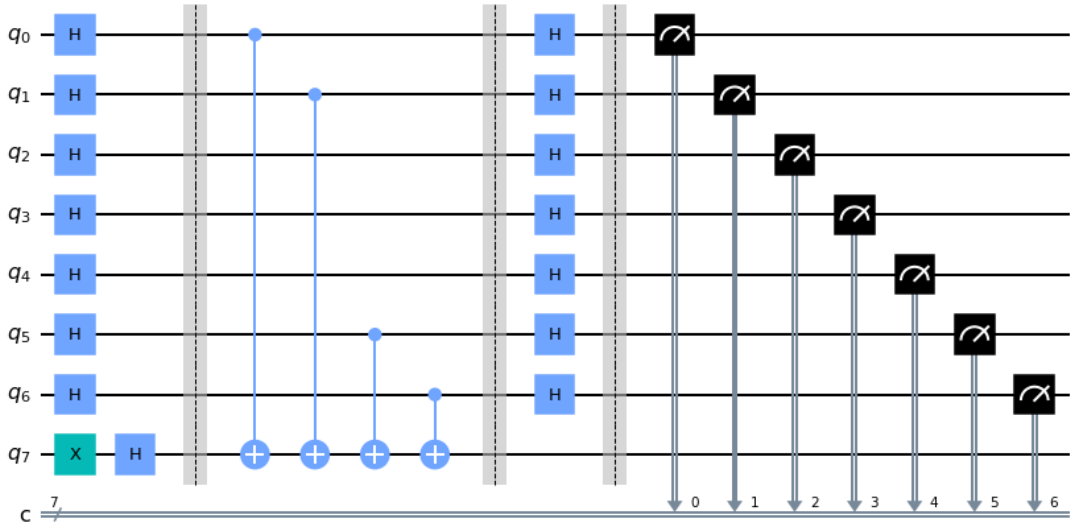
[31]:
```
SecretNumber = '1100011'
#izberemo si neko skrivno število, ki jo računalnik mora ugotoviti
```

[32]:
```
circuit = QuantumCircuit(len(SecretNumber)+1, len(SecretNumber))
#določimo dolžino število kvantnih in klasičanih registrov
circuit.h(range(len(SecretNumber)))
circuit.x(len(SecretNumber))
circuit.h(len(SecretNumber))
#določimo na kateri qubit postavimo h in not (x) vrata glede na dolžino števila
circuit.barrier()

for x, y in enumerate(reversed(SecretNumber)):
    if y == '1':
        circuit.cx(x, len(SecretNumber))
#določimo kje bomo postavili kontrolna not vrata glede na vrednosti skrivne
 ↪številke (na katerih mesti so enice)

circuit.barrier()

circuit.h(range(len(SecretNumber)))
#dodamo h vrata na vse qubite razen zadnjega v krogu
circuit.barrier()
circuit.measure(range(len(SecretNumber)), range(len(SecretNumber)))
#izmerimo stanje qubitov
```

[32]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x7efca500a380>
```

[33]:
```
circuit.draw(output='mpl')
```

[33]:



[34]:
```
simulator = Aer.get_backend('qasm_simulator')
#določimo simulator
result = execute(circuit, backend = simulator, shots = 1).result()
counts = result.get_counts()
print(counts)
```

```
{'1100011': 1}
```

Skrivno število smo pravilno ugotovili že v prvem poskusu, najdemo pa ga glede na postavitev c-n
vrat v krogu.

Priloga 2

kvantna teleportacija

March 5, 2022

Vemo da qubitov ne moremo kopirati, saj ne poznamo njihovega stanja, z izmeritvijo pa bi stanje razpadlo. Z algoritmom kvantne teleportacije preslikamo vrednost enega qubita v vrednost klasičnega bita, ki ga lahko potem izmerimo. Sestavo kroga bomo naredili kot iz naloge, kjer je že narejena sestava kroga in razlaga.

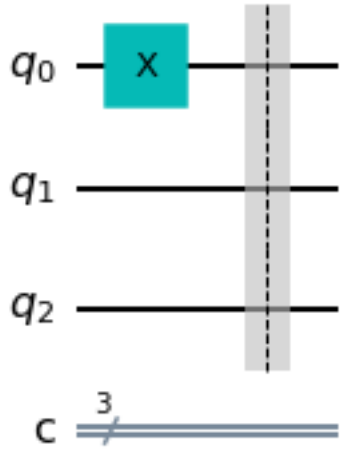
[1]:
```
from qiskit import *
%matplotlib inline
from qiskit import IBMQ
IBMQ.load_account()
```

[1]:
```
<AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>
```

[2]:
```
circuit = QuantumCircuit(3, 3)
#določimo kvantni krog s 3 qubiti in 3 klasičnimi registri
```

[3]:
```
circuit.x(0)
circuit.barrier()
circuit.draw(output='mpl')
#dodamo not vrata na qubit katerega vrednost želimo teleportirati
```

[3]:

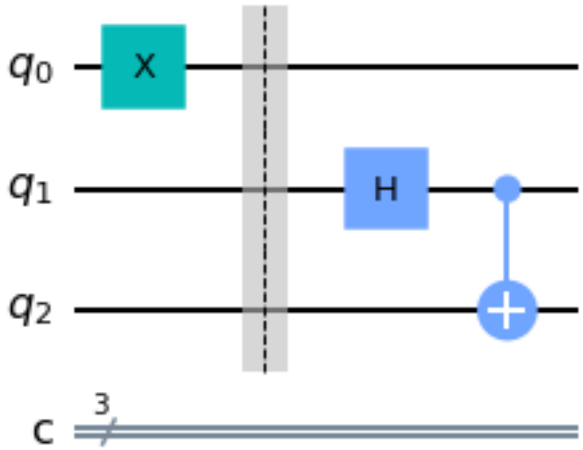


[4]:
```
circuit.h(1)
circuit.cx(1, 2)
```

[4]: <qiskit.circuit.instructionset.InstructionSet at 0x7f10cb0d06c0>

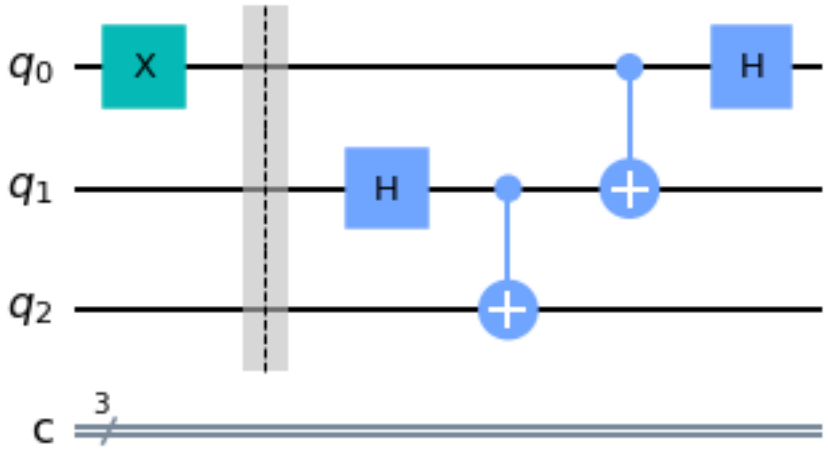
[5]:
```
circuit.draw(output='mpl')
```

[5]:



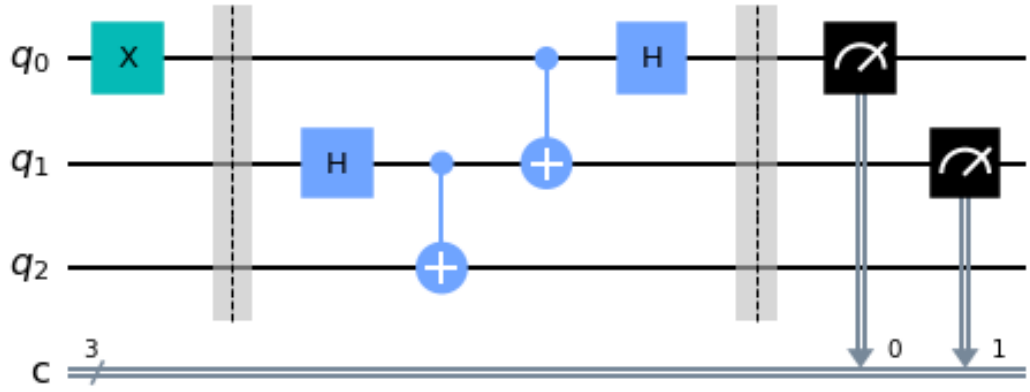
[6]:
```
circuit.cx(0,1)
circuit.h(0)
circuit.draw(output='mpl')
```

[6]:



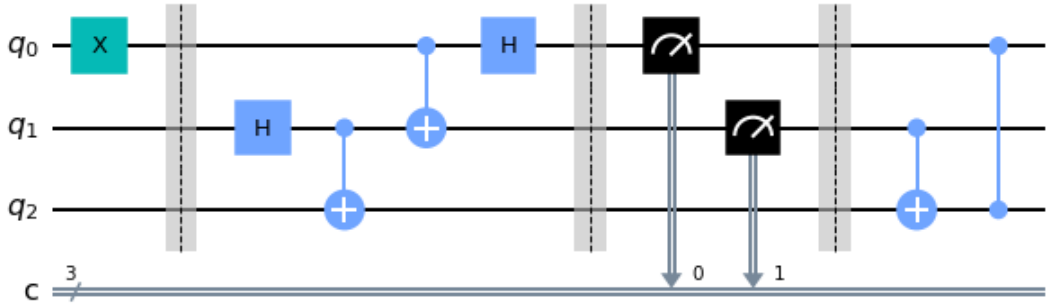
[7]:
```
circuit.barrier()
circuit.measure([0,1], [0,1])
circuit.draw(output='mpl')
```

[7]:



[8]:
```
circuit.barrier()
circuit.cx(1,2)
circuit.cz(0,2)
circuit.draw(output='mpl')
```

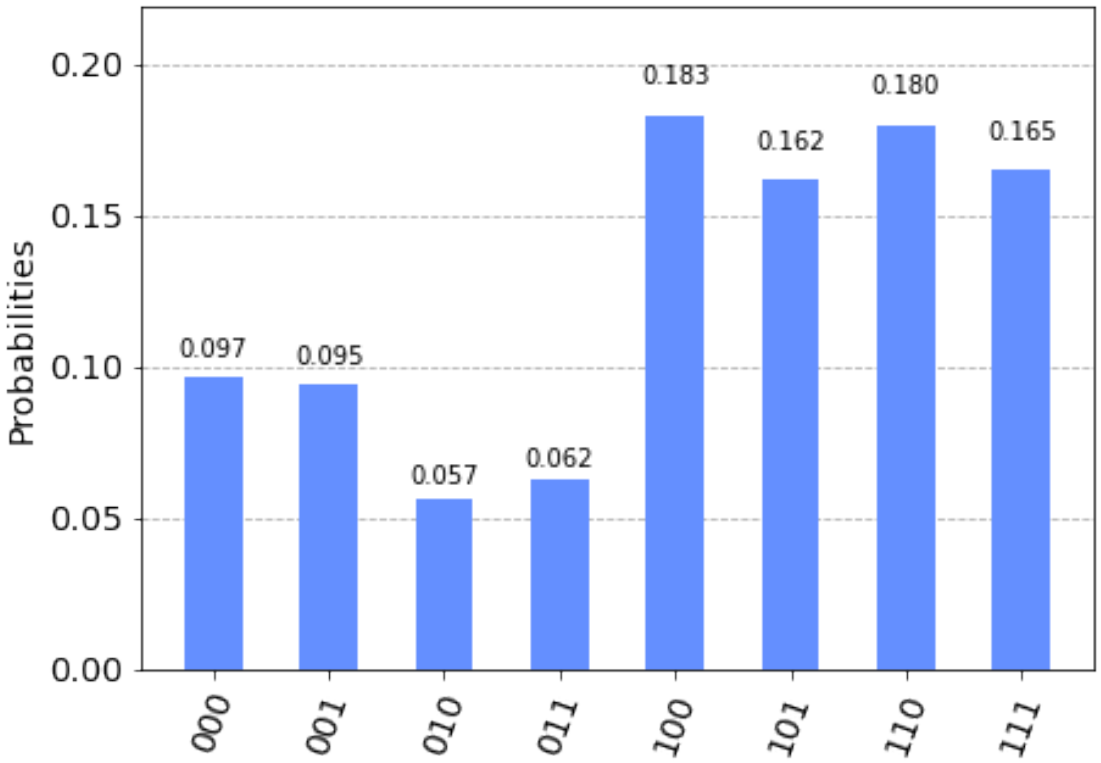
[8]:



[10]:
```
circuit.measure(2,2)
provider = IBMQ.get_provider()
simulator = provider.get_backend('ibmq_manila')
rezultat = execute(circuit, backend = simulator, shots = 1024).result()
counts = rezultat.get_counts()
from qiskit.tools.visualization import plot_histogram
```

```
plot_histogram(counts)
#v tem koraku definiramo simulator in izvršimo krog. Nato še prikažemo
 ↪rezultati z grafi
```

[10]:



[11]:
```
print(counts)
```

```
{'000': 99, '001': 97, '010': 58, '011': 64, '100': 187, '101': 166, '110': 184,
'111': 169}
```

Končni grafi kažejo vrednosti klasičnih bitov od c2 do c0. Ker smo teleportirali vrednost 1 iz qubita 2 v bit 2 vidimo, da imajo vse vrednosti v klasičnem bitu 2 vrednost 1.

Priloga 3

Groverjev algoritem

March 5, 2022

najprej prikažimo kako bi klasični računalnik iskal število in koliko poskusov bi klasični računalnik lahko s Brute-force (poskušanjem vsake možnosti) pridejo do rešitve, ampak za to potrebujejo toliko poskusov, na katerem indexu je bila vrednost +1.

Klasično

[16]:
```
moj_list=[5,1,9,1,4,4,7,1,9,5,6]
#naredimo si svoj list z naključnimi števili
```

[17]:
```
def oracle(moj_input):
#definiramo novo funkcijo oracle (gre za oracle algoritem)
    iskano=6
#določimo iskano število
    if moj_input is iskano:
        odgovor = True
    else:
        odgovor = False
#določimo iskanje v našem inputu
    return odgovor
```

[18]:
```
for index, poskus in enumerate(moj_list):
#določimo na katerem index najdemo število in ali število ustreza iskanemu
    if oracle(poskus) is True:
        print('iskano število najdeno na index %i'%index)
        print('%i poskusov porabljenih'%(index+1))
        break
```

```
iskano število najdeno na index 10
11 poskusov porabljenih
```

[19]:
```
from qiskit import *
import matplotlib.pyplot as plt
import numpy as np
from qiskit import IBMQ
IBMQ.load_account()
#uvozimo knjižnice qiskit, matplot za prikaz in numpy za števila
```

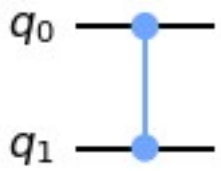
```
ibmqfactory.load_account:WARNING:2022-03-05 14:35:50,220: Credentials are
already in use. The existing account in the session will be replaced.
```

[19]: <AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>

Zdaj sestavimo oracle, ki bo preveril vsak input in vrednosti ki je pravilna spremenil fazo (predznal
Oracle ni univerzalen, tako da vsakič ko želimo spremeniti vrednost iskanega števila, moramo
vrednosti oracla prilagotiti.

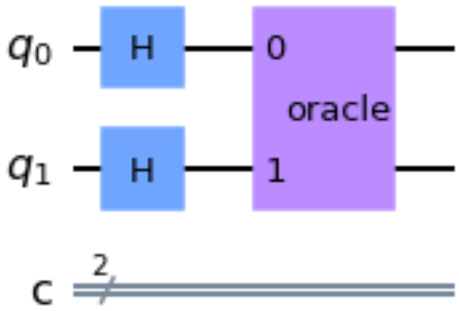
[20]:
```
oracle = QuantumCircuit(2,name='oracle')
oracle.cz(0,1)
oracle.to_gate()
oracle.draw(output='mpl')
#definiramo krog oracle (postavimo control-z vrata) in krog spremenimo v vrata.
```

[20]:



[21]:
```
backend = Aer.get_backend('statevector_simulator')
#določimo simulator
grover = QuantumCircuit(2,2)
grover.h([0,1])
#naredimo groverjev krog in določimo vrata na qubita 0 in 1
grover.append(oracle,[0,1])
#dodamo oracle del v krog
grover.draw(output='mpl')
```

[21]:



[22]:
```
izvedba = execute(grover,backend)
rezultat = izvedba.result()
#izvedemo krog
```

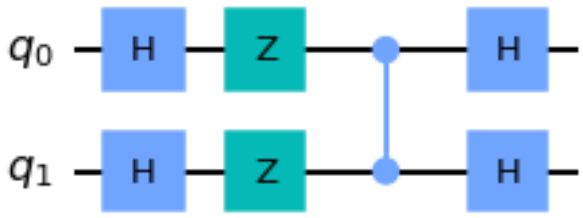
[23]:
```
sv = rezultat.get_statevector()
np.around(sv,2)
#predstavimo rezultate kot vektorje in jih
```

[23]: array([0.5+0.j, 0.5+0.j, 0.5+0.j, -0.5+0.j])

V rezultatih vidimo da naš oracle deluje, saj je na mestu 3 (11 – Naš oracle deluje tako, da če stanje prepozna kot iskano, spremeni fazo stanja (spremeni predznak). Kot pa kvadrat vektorja predstavi verjetnost določene vrednost, negativen vektor ne more predstljati verjetnosti. Zato moramo narediti drugo komponento "Reflection" zrcaljenje operator. V tem s procesom amplitudne amplifikacije povečamo možnosti iskanega stanja in zmanjšamo možnosti za vse ostala. Zrcaljenje mu reče zato, ker v 2d ravnini izgleda kot da smo stanje S (ki predstavlja superpozicijo) prezrcalili čez stanje S', ki predstavlja vektor S z odštetim vektorjem W (ki pa predstavlja iskano vrednost). Podrobno razlago pa bom zapisal v nalogi ob skicah in grafih.

[24]:
```
zrcaljenje = QuantumCircuit(2,name='zrcaljenje')
zrcaljenje.h([0,1])
zrcaljenje.z([0,1])
zrcaljenje.cz(0,1)
zrcaljenje.h([0,1])
zrcaljenje.to_gate()
zrcaljenje.draw(output='mpl')
```

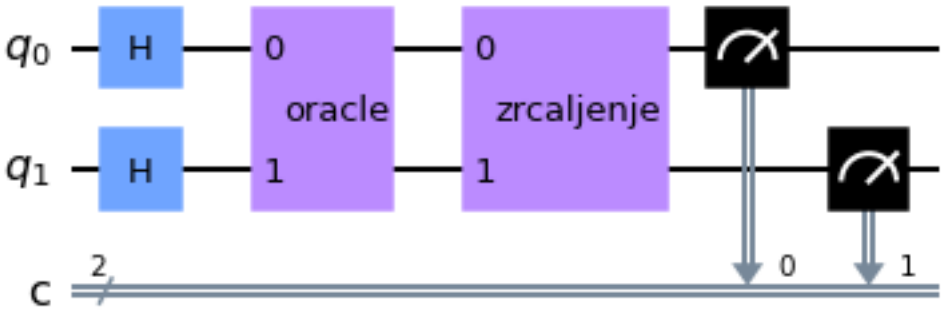
[24]:



[25]:
```
provider = IBMQ.get_provider()
backend = provider.get_backend('ibmq_manila')
grover = QuantumCircuit(2,2)
grover.h([0,1])
grover.append(oracle,[0,1])
grover.append(zrcaljenje,[0,1])
grover.measure([0,1],[0,1])
```

```
grover.draw(output='mpl')
#na konču še naredimo končen krog, ki izvaja Groverjev algoritem. Dodamo h
 ↪vrata in vrata ki smo jih naredili: oracle in zrcaljenje
```

[25]:



[26]:
```
izvedba = execute(grover,backend,shots=1)
rezultati = izvedba.result()
rezultati.get_counts()
```

[26]: {'11': 1}

Kot vidimo smo stanje 11, iskano stanje, našli že v prvem poskusu. S povečanim krogom in prilagoditvijo oracla bi lahko iskali vrednosti tudi za več kot pa samo 4 vrednosti oz. 2 qubita.

Priloga 4

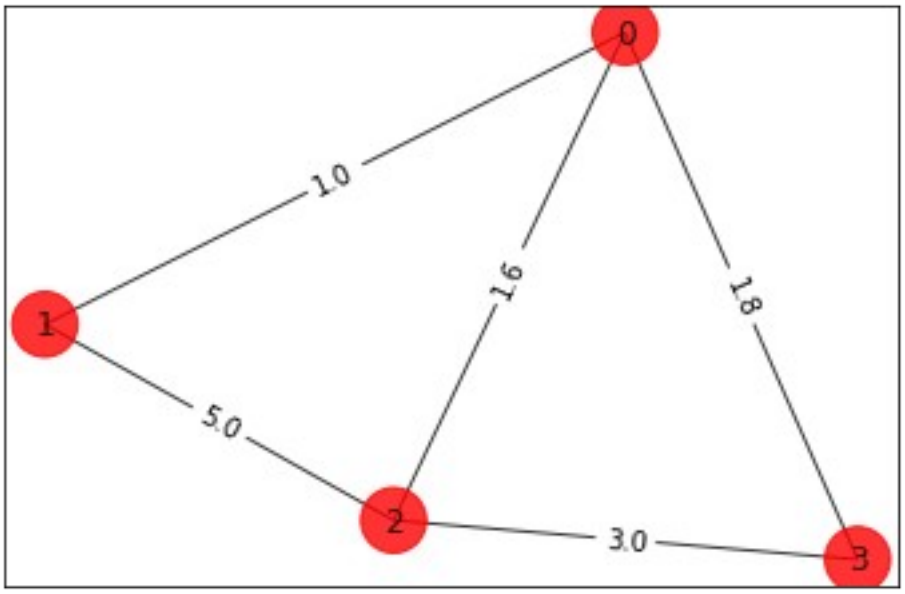
Max-Cut kvantna rešitev

March 5, 2022

[50]:
```
import matplotlib.pyplot as plt
import matplotlib.axes as axes
import numpy as np
import networkx as nx

from qiskit import Aer
from qiskit.tools.visualization import plot_histogram
from qiskit.circuit.library import TwoLocal
from qiskit_optimization.applications import Maxcut, Tsp
from qiskit.algorithms import VQE, NumPyMinimumEigensolver
from qiskit.algorithms.optimizers import SPSA
from qiskit.utils import algorithm_globals, QuantumInstance
from qiskit_optimization.algorithms import MinimumEigenOptimizer
from qiskit_optimization.problems import QuadraticProgram
import timeit
starttime = timeit.default_timer()
```

[51]:
```
# naredimo graf. Določimo število točk, uteži točk, definiramo funkcijo za prihodnje risanje grafov

n = 4 # Number of nodes in graph
G = nx.Graph()
G.add_nodes_from(np.arange(0, n, 1))
elist = [(0, 1, 1.0), (0, 2, 1.6), (0, 3, 1.8), (1, 2, 5.0), (2, 3, 3.0)]
G.add_weighted_edges_from(elist)

colors = ["r" for node in G.nodes()]
pos = nx.spring_layout(G)


def draw_graph(G, colors, pos):
    default_axes = plt.axes(frameon=True)
    nx.draw_networkx(G, node_color=colors, node_size=600, alpha=0.8, ax=default_axes, pos=pos)
    edge_labels = nx.get_edge_attributes(G, "weight")
    nx.draw_networkx_edge_labels(G, pos=pos, edge_labels=edge_labels)
```

```
draw_graph(G, colors, pos)
```



[52]:
```
# naredimo matrico iz uteži posameznih točk za prenos grafa v matrico.
matrica = np.zeros([n,n])
for i in range(n):
    for j in range(n):
        temp = G.get_edge_data(i,j,default=0)
        if temp != 0:
            matrica[i,j] = temp['weight']
print(matrica)
```

```
[[0. 1. 1.6 1.8]
 [1. 0. 5. 0. ]
 [1.6 5. 0. 3. ]
 [1.8 0. 3. 0. ]]
```

[53]:
```
# spremenimo matrico (graf) v kvadratni program za računanje s kvantnim
 ↪računalnikom.
max_cut = Maxcut(matrica)
kvadratni = max_cut.to_quadratic_program()
print(kvadratni.export_as_lp_string())
```

```
\ This file has been generated by DOcplex
\ ENCODING=ISO-8859-1
\Problem name: Max-cut
```

```
Maximize
 obj: 4.400000000000 x_0 + 6 x_1 + 9.600000000000 x_2 + 4.800000000000 x_3 + [
      - 4 x_0*x_1 - 6.400000000000 x_0*x_2 - 7.200000000000 x_0*x_3 - 20 x_1*x_2
      - 12 x_2*x_3 ]/2
Subject To

Bounds
 0 <= x_0 <= 1
 0 <= x_1 <= 1
 0 <= x_2 <= 1
 0 <= x_3 <= 1

Binaries
 x_0 x_1 x_2 x_3
End
```

[54]:
```
# določimo ising hamiltonian
qubitOp, offset = kvadratni.to_ising()
print("Offset:", offset)
print("Ising Hamiltonian:")
print(str(qubitOp))
```

```
Offset: -6.199999999999999
Ising Hamiltonian:
1.5 * ZZII
+ 2.5 * IZZI
+ 0.9 * ZIIZ
+ 0.8 * IZIZ
+ 0.5 * IIZZ
```

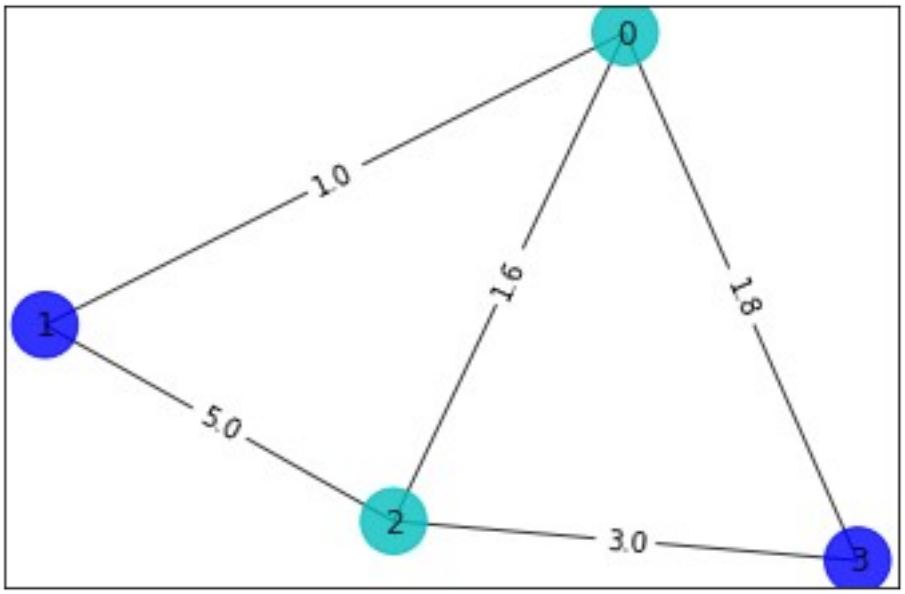
[55]:
```
exact = MinimumEigenOptimizer(NumPyMinimumEigensolver())
result = exact.solve(kvadratni)
print(result)
```

```
optimal function value: 10.8
optimal value: [1. 0. 1. 0.]
status: SUCCESS
```

[56]:
```
ee = NumPyMinimumEigensolver()
result = ee.compute_minimum_eigenvalue(qubitOp)

x = max_cut.sample_most_likely(result.eigenstate)
print("energija:", result.eigenvalue.real)
print("max-cut cilj:", result.eigenvalue.real + offset)
print("rešitev:", x)
print("ciljana rešitev:", kvadratni.objective.evaluate(x))
```

```
colors = ["b" if x[i] == 0 else "c" for i in range(n)]
draw_graph(G, colors, pos)
```

```
energija: -4.6000000000000005
max-cut cilj: -10.8
rešitev: [1 0 1 0]
ciljana rešitev: 10.8
```

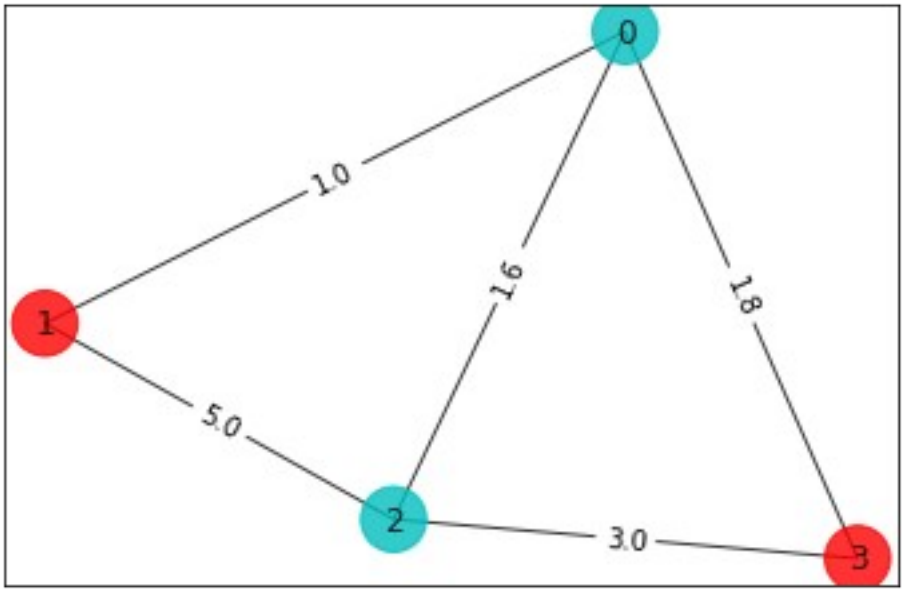


[57]:
```
# določimo simulator in število poskusov na simulatorju
algorithm_globals.random_seed = 123
seed = 10598
backend = Aer.get_backend("aer_simulator_statevector")
quantum_instance = QuantumInstance(backend, seed_simulator=seed,␣
↪seed_transpiler=seed)
```

[58]:
```
# naredimo VQE algoritem za izvedbo na kvantnem računalniku
spsa = SPSA(maxiter=300)
ry = TwoLocal(qubitOp.num_qubits, "ry", "cz", reps=5, entanglement="linear")
vqe = VQE(ry, optimizer=spsa, quantum_instance=quantum_instance)

# zaženemo
result = vqe.compute_minimum_eigenvalue(qubitOp)

x = max_cut.sample_most_likely(result.eigenstate)
print("energija:", result.eigenvalue.real)
print("čas:", result.optimizer_time)
print("max-cut cilj:", result.eigenvalue.real + offset)
```

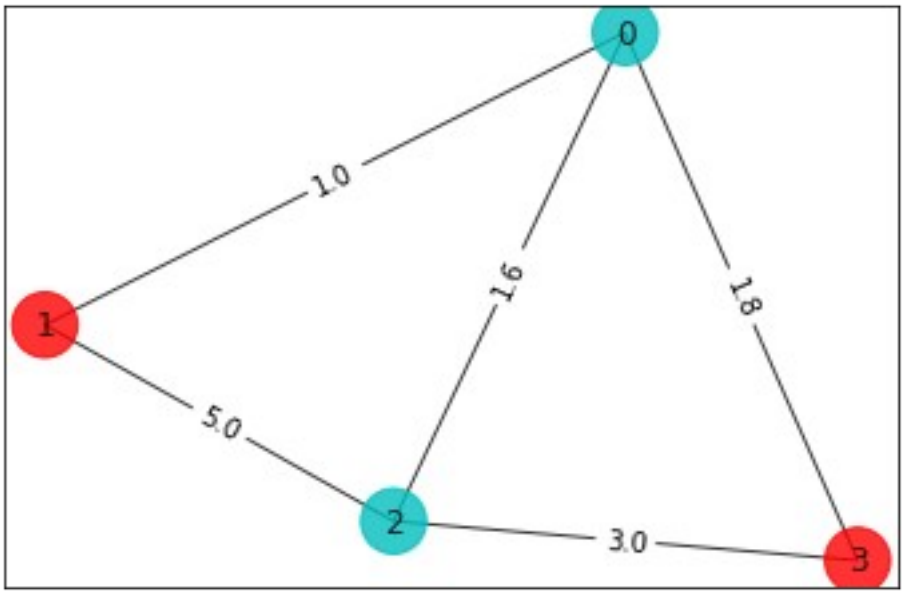
```
print("rešitev:", x)
print("ciljana rešitev:", kvadratni.objective.evaluate(x))

colors = ["r" if x[i] == 0 else "c" for i in range(n)]
draw_graph(G, colors, pos)
```

```
energija: -4.598737546191859
čas: 2.4575252532958984
max-cut cilj: -10.798737546191859
rešitev: [1. 0. 1. 0.]
ciljana rešitev: 10.8
```



[59]:
```
# naredimo optimizator iz VQE
vqe_optimizer = MinimumEigenOptimizer(vqe)

# rešimo kvadratni program
result = vqe_optimizer.solve(kvadratni)
print(result)

colors = ['r' if result.x[i] == 0 else 'c' for i in range(n)]
draw_graph(G, colors, pos)
```

```
optimal function value: 10.8
optimal value: [1. 0. 1. 0.]
status: SUCCESS
```



[60]:
```
print("čas:", timeit.default_timer() - starttime)
```

```
čas: 5.406315835003625
```

Priloga 4a

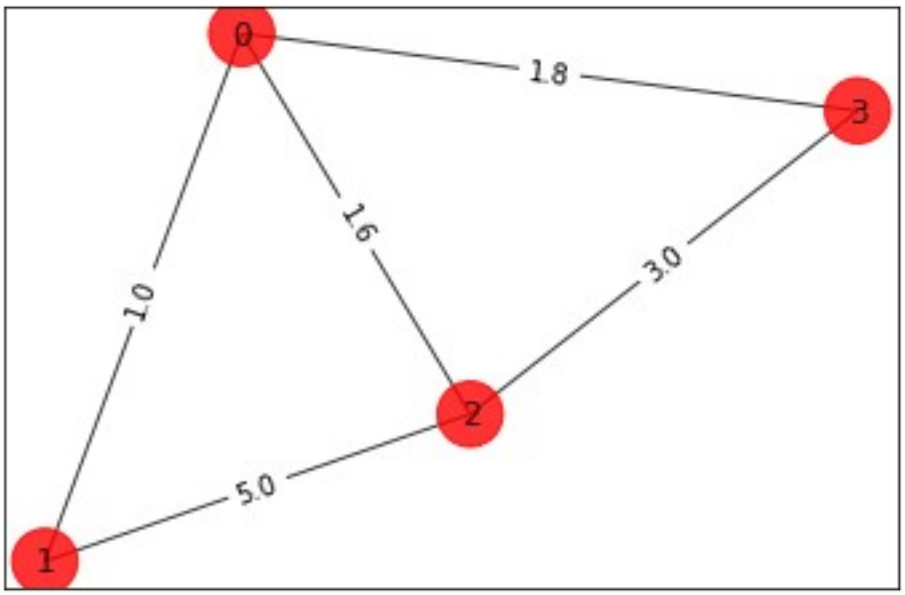
Max-cut surova sila

March 5, 2022

[50]:
```
import matplotlib.pyplot as plt
import matplotlib.axes as axes
import numpy as np
import networkx as nx

from qiskit import Aer
from qiskit.tools.visualization import plot_histogram
from qiskit.circuit.library import TwoLocal
from qiskit_optimization.applications import Maxcut, Tsp
from qiskit.algorithms import VQE, NumPyMinimumEigensolver
from qiskit.algorithms.optimizers import SPSA
from qiskit.utils import algorithm_globals, QuantumInstance
from qiskit_optimization.algorithms import MinimumEigenOptimizer
from qiskit_optimization.problems import QuadraticProgram
import timeit
starttime = timeit.default_timer()
```

[51]:
```
#naredimo graf

n = 4 # število točk
G = nx.Graph()
G.add_nodes_from(np.arange(0, n, 1))
elist = [(0, 1, 1.0), (0, 2, 1.6), (0, 3, 1.8), (1, 2, 5.0), (2, 3, 3.0)]
G.add_weighted_edges_from(elist)

colors = ["r" for node in G.nodes()]
pos = nx.spring_layout(G)


def draw_graph(G, colors, pos):
    default_axes = plt.axes(frameon=True)
    nx.draw_networkx(G, node_color=colors, node_size=600, alpha=0.8,␣
 ↪ax=default_axes, pos=pos)
    edge_labels = nx.get_edge_attributes(G, "weight")
    nx.draw_networkx_edge_labels(G, pos=pos, edge_labels=edge_labels)
```

```
draw_graph(G, colors, pos)
```



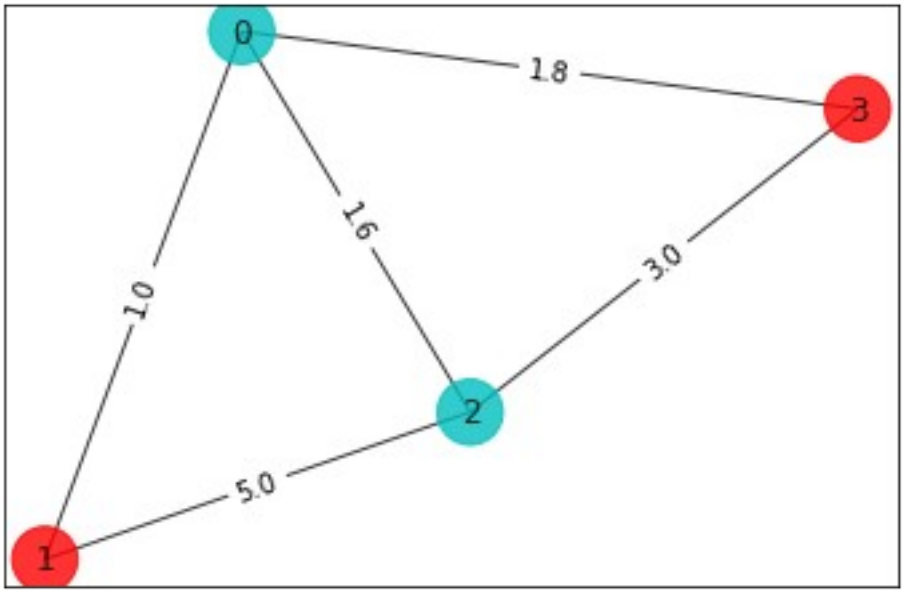
[52]:
```
# naredimo matrico
w = np.zeros([n, n])
for i in range(n):
    for j in range(n):
        temp = G.get_edge_data(i, j, default=0)
        if temp != 0:
            w[i, j] = temp["weight"]
print(w)
```

```
[[0. 1. 1.6 1.8]
 [1. 0. 5. 0. ]
 [1.6 5. 0. 3. ]
 [1.8 0. 3. 0. ]]
```

[53]:
```
# napišemo program, kjer preizkusimo vsako možnost do najboljše
best_cost_brute = 0
for b in range(2 ** n):
    x = [int(t) for t in reversed(list(bin(b)[2:].zfill(n)))]
    cost = 0
    for i in range(n):
        for j in range(n):
            cost = cost + w[i, j] * x[i] * (1 - x[j])
    if best_cost_brute < cost:
        best_cost_brute = cost
        xbest_brute = x
```

```
    print("primer:" + str(x) + " cena: " + str(cost))

colors = ["r" if xbest_brute[i] == 0 else "c" for i in range(n)]
draw_graph(G, colors, pos)
print("\nNajboljša rešitev: " + str(xbest_brute) + " cena: " +␣
↪str(best_cost_brute))
```

```
primer:[0, 0, 0, 0] cena: 0.0
primer:[1, 0, 0, 0] cena: 4.4
primer:[0, 1, 0, 0] cena: 6.0
primer:[1, 1, 0, 0] cena: 8.4
primer:[0, 0, 1, 0] cena: 9.6
primer:[1, 0, 1, 0] cena: 10.8
primer:[0, 1, 1, 0] cena: 5.6
primer:[1, 1, 1, 0] cena: 4.8
primer:[0, 0, 0, 1] cena: 4.8
primer:[1, 0, 0, 1] cena: 5.6
primer:[0, 1, 0, 1] cena: 10.8
primer:[1, 1, 0, 1] cena: 9.6
primer:[0, 0, 1, 1] cena: 8.4
primer:[1, 0, 1, 1] cena: 6.0
primer:[0, 1, 1, 1] cena: 4.4
primer:[1, 1, 1, 1] cena: 0.0

Najboljša rešitev: [1, 0, 1, 0] cena: 10.8
```



[54]:
```
print("čas:", timeit.default_timer() - starttime)
```

```
čas: 0.20321389700256987
```

[ ]:

Priloga 5

Problem potujočega trgovca-kvantna rešitev

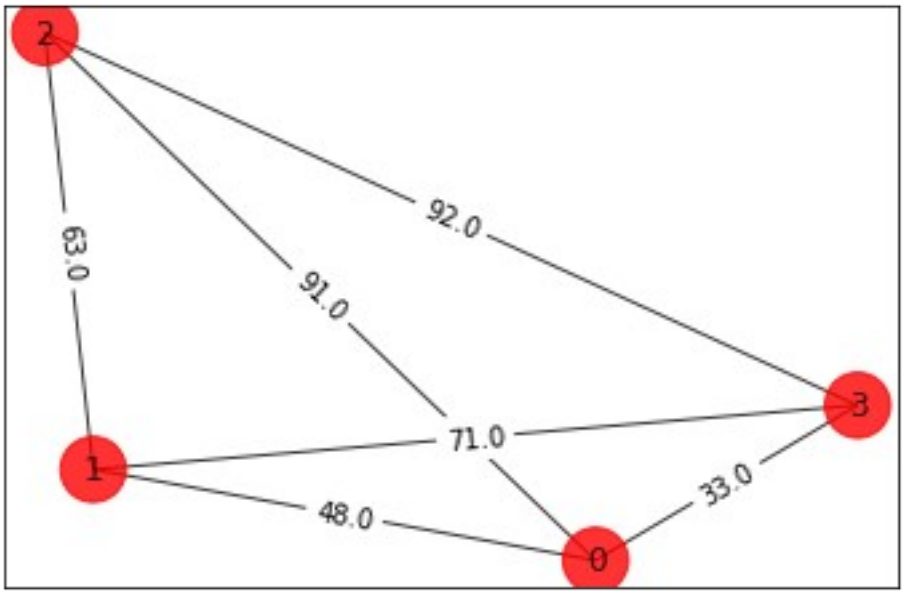
March 6, 2022

[33]:
```
import matplotlib.pyplot as plt
import matplotlib.axes as axes
import numpy as np
import networkx as nx

from qiskit import Aer
from qiskit.tools.visualization import plot_histogram
from qiskit.circuit.library import TwoLocal
from qiskit_optimization.applications import Maxcut, Tsp
from qiskit.algorithms import VQE, NumPyMinimumEigensolver
from qiskit.algorithms.optimizers import SPSA
from qiskit.utils import algorithm_globals, QuantumInstance
from qiskit_optimization.algorithms import MinimumEigenOptimizer
from qiskit_optimization.problems import QuadraticProgram
import timeit
starttime = timeit.default_timer()
```

[34]:
```
# določimo graf in postavitev za problem
def draw_graph(G, colors, pos):
    default_axes = plt.axes(frameon=True)
    nx.draw_networkx(G, node_color=colors, node_size=600, alpha=0.8,␣
 ↪ax=default_axes, pos=pos)
    edge_labels = nx.get_edge_attributes(G, "weight")
    nx.draw_networkx_edge_labels(G, pos=pos, edge_labels=edge_labels)

n = 4
num_qubits = n ** 2
tsp = Tsp.create_random_instance(n, seed=123)
adj_matrix = nx.to_numpy_matrix(tsp.graph)
print("razdalje\n", adj_matrix)

colors = ["r" for node in tsp.graph.nodes]
pos = [tsp.graph.nodes[node]["pos"] for node in tsp.graph.nodes]
draw_graph(tsp.graph, colors, pos)
```

```
razdalje
 [[ 0. 48. 91. 33.]
 [48.  0. 63. 71.]
```

```
[91. 63. 0. 92.]
[33. 71. 92.0.]]
```



[35]:
```
exact = MinimumEigenOptimizer(NumPyMinimumEigensolver())
```

[36]:
```
qp = tsp.to_quadratic_program()
```

[37]:
```
# Problem pretvorimo
from qiskit_optimization.converters import QuadraticProgramToQubo

qp2qubo = QuadraticProgramToQubo()
qubo = qp2qubo.convert(qp)
qubitOp, offset = qubo.to_ising()
```

[38]:
```
result = exact.solve(qubo)
print(result)
```

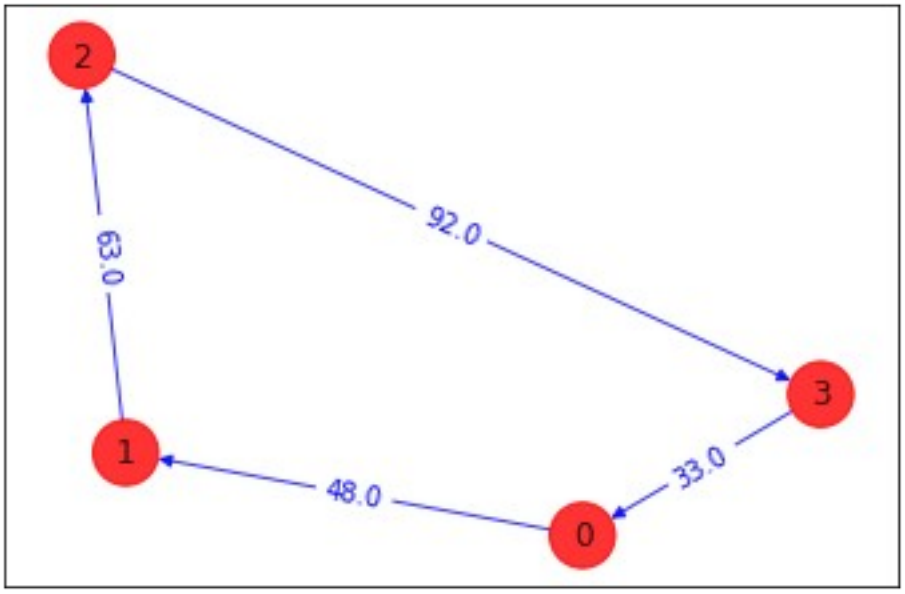
```
optimal function value: 236.0
optimal value: [0. 0. 0. 1. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0.]
status: SUCCESS
```

[39]:
```
# definiramo funkcijo za risanje grafov
def draw_tsp_solution(G, order, colors, pos):
    G2 = nx.DiGraph()
    G2.add_nodes_from(G)
    n = len(order)
    for i in range(n):
```

```
        j = (i + 1) % n
        G2.add_edge(order[i], order[j], weight=G[order[i]][order[j]]["weight"])
    default_axes = plt.axes(frameon=True)
    nx.draw_networkx(
        G2, node_color=colors, edge_color="b", node_size=600, alpha=0.8,␣
↪ax=default_axes, pos=pos
    )
    edge_labels = nx.get_edge_attributes(G2, "weight")
    nx.draw_networkx_edge_labels(G2, pos, font_color="b",␣
↪edge_labels=edge_labels)
```

[40]:
```
ee = NumPyMinimumEigensolver()
result = ee.compute_minimum_eigenvalue(qubitOp)

print("energija:", result.eigenvalue.real)
print("iskano:", result.eigenvalue.real + offset)
x = tsp.sample_most_likely(result.eigenstate)
print("feasible:", qubo.is_feasible(x))
z = tsp.interpret(x)
print("rešitev:", z)
print("iskana rešitev:", tsp.tsp_value(z, adj_matrix))
draw_tsp_solution(tsp.graph, z, colors, pos)
```

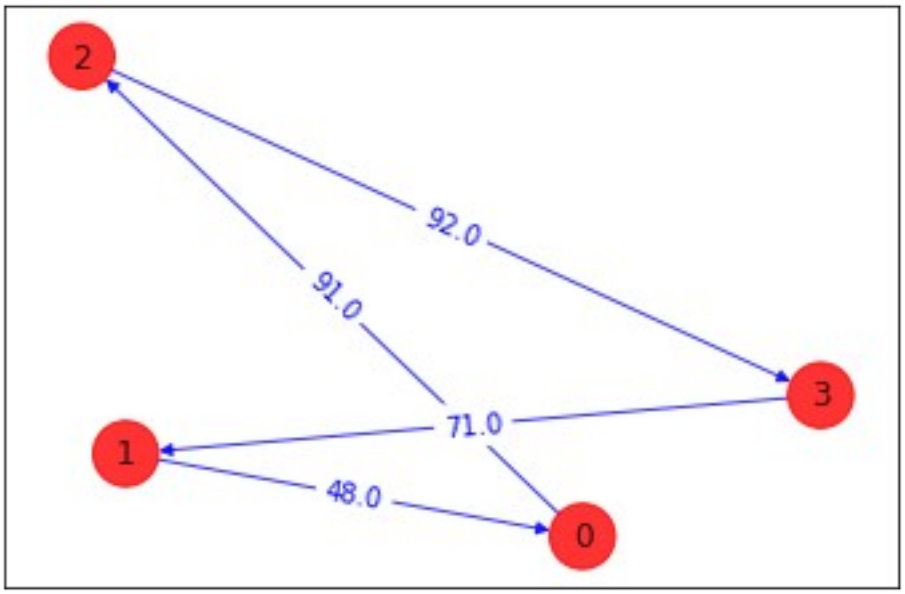
```
energija: -51520.0
iskano: 236.0
feasible: True
rešitev: [1, 2, 3, 0]
iskana rešitev: 236.0
```



[31]:
```
# problem še zaženemo na kvantnem računalniku
algorithm_globals.random_seed = 123
seed = 10598
backend = Aer.get_backend("qasm_simulator")
quantum_instance = QuantumInstance(backend, seed_simulator=seed,␣
↪seed_transpiler=seed)
```

[32]:
```
spsa = SPSA(maxiter=300)
ry = TwoLocal(qubitOp.num_qubits, "ry", "cz", reps=5, entanglement="linear")
vqe = VQE(ry, optimizer=spsa, quantum_instance=quantum_instance)

result = vqe.compute_minimum_eigenvalue(qubitOp)

print("energy:", result.eigenvalue.real)
print("time:", result.optimizer_time)
x = tsp.sample_most_likely(result.eigenstate)
print("feasible:", qubo.is_feasible(x))
z = tsp.interpret(x)
print("solution:", z)
print("solution objective:", tsp.tsp_value(z, adj_matrix))
draw_tsp_solution(tsp.graph, z, colors, pos)
```

```
energy: -48489.29199218752
time: 417.0443105697632
feasible: True
solution: [3, 1, 0, 2]
solution objective: 302.0
```



Priloga 5a

Problem potujočega trgovca-rešitev s surovo silo

March 6, 2022

[1]:
```
import matplotlib.pyplot as plt
import matplotlib.axes as axes
import numpy as np
import networkx as nx
import timeit

from qiskit import Aer
from qiskit.tools.visualization import plot_histogram
from qiskit.circuit.library import TwoLocal
from qiskit_optimization.applications import Maxcut, Tsp
from qiskit.algorithms import VQE, NumPyMinimumEigensolver
from qiskit.algorithms.optimizers import SPSA
from qiskit.utils import algorithm_globals, QuantumInstance
from qiskit_optimization.algorithms import MinimumEigenOptimizer
from qiskit_optimization.problems import QuadraticProgram
```

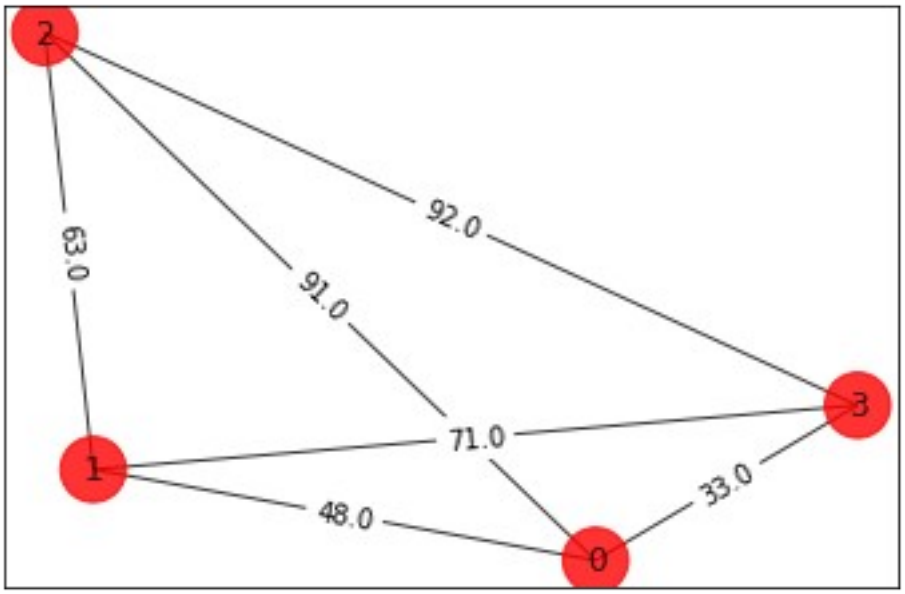
[2]:
```
starttime = timeit.default_timer()
```

[3]:
```
def draw_graph(G, colors, pos):
    default_axes = plt.axes(frameon=True)
    nx.draw_networkx(G, node_color=colors, node_size=600, alpha=0.8,␣
 ↪ax=default_axes, pos=pos)
    edge_labels = nx.get_edge_attributes(G, "weight")
    nx.draw_networkx_edge_labels(G, pos=pos, edge_labels=edge_labels)
```

[4]:
```
n = 4
num_qubits = n ** 2
tsp = Tsp.create_random_instance(n, seed=123)
adj_matrix = nx.to_numpy_matrix(tsp.graph)
print("distance\n", adj_matrix)

colors = ["r" for node in tsp.graph.nodes]
pos = [tsp.graph.nodes[node]["pos"] for node in tsp.graph.nodes]
draw_graph(tsp.graph, colors, pos)
```

```
distance
 [[ 0. 48. 91. 33.]
 [48.  0. 63. 71.]
```

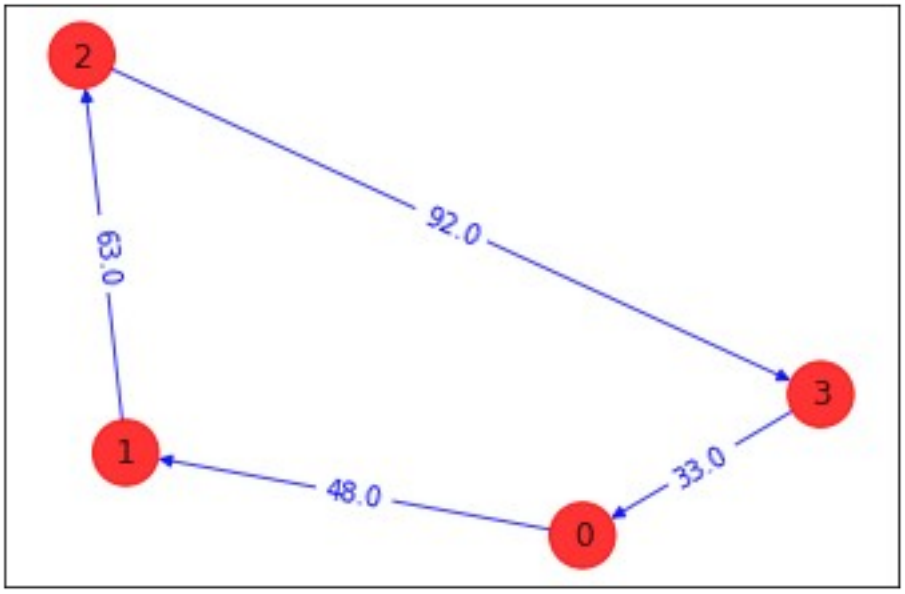
```
[91. 63. 0. 92.]
[33. 71. 92.0.]]
```



[5]:
```python
from itertools import permutations


def brute_force_tsp(w, N):
    a = list(permutations(range(1, N)))
    last_best_distance = 1e10
    for i in a:
        distance = 0
        pre_j = 0
        for j in i:
            distance = distance + w[j, pre_j]
            pre_j = j
        distance = distance + w[pre_j, 0]
        order = (0,) + i
        if distance < last_best_distance:
            best_order = order
            last_best_distance = distance
            print("order = " + str(order) + " Distance = " + str(distance))
    return last_best_distance, best_order


best_distance, best_order = brute_force_tsp(adj_matrix, n)
print(
    "Best order from brute force = "
```

```
    + str(best_order)
    + " with total distance = "
    + str(best_distance)
)


def draw_tsp_solution(G, order, colors, pos):
    G2 = nx.DiGraph()
    G2.add_nodes_from(G)
    n = len(order)
    for i in range(n):
        j = (i + 1) % n
        G2.add_edge(order[i], order[j], weight=G[order[i]][order[j]]["weight"])
    default_axes = plt.axes(frameon=True)
    nx.draw_networkx(
        G2, node_color=colors, edge_color="b", node_size=600, alpha=0.8,␣
 ↪ax=default_axes, pos=pos
    )
    edge_labels = nx.get_edge_attributes(G2, "weight")
    nx.draw_networkx_edge_labels(G2, pos, font_color="b",␣
 ↪edge_labels=edge_labels)


draw_tsp_solution(tsp.graph, best_order, colors, pos)
```

```
order = (0, 1, 2, 3) Distance = 236.0
Best order from brute force = (0, 1, 2, 3) with total distance = 236.0
```



[6]:
```
print("čas :", timeit.default_timer() - starttime)
```

```
čas : 0.2771052919997601
```

[ ]:
```

```

Priloga 5b

Problem potujočega trgovca-dinamično programiranje

March 6, 2022

[15]:
```
import random
import timeit
def SetCostMatrix(num):
    cmatrix = {}
    for i in range(1, num + 1):
        for j in range(1, num + 1):
            if i == j:
                cmatrix[(i, j)] = 0
            else:
                cmatrix[(i, j)] = random.randint(10, 50)
    return cmatrix
starttime = timeit.default_timer()
```

[16]:
```
total_num = 4
num_cities = SetCostMatrix(total_num)
```

[17]:
```
def GetCostVal(row, col, source):
    if col == 0:
        col = source
        return num_cities[(row, col)]
    return num_cities[(row, col)]
```

[18]:
```
iterative_process = []
def TSPGetMinDistance(main_source, source, cities):
    if len(cities) == 1:
        minDis = GetCostVal(source, cities[0], main_source) +␣
↪GetCostVal(cities[0], 0, main_source)
        return minDis
    else:
        Dist = []
        for city in cities:
            dcities = cities[:]
            dcities.remove(city)
            Dist.append(GetCostVal(source, city, source) +␣
↪TSPGetMinDistance(main_source, city, dcities))
        iterative_process.append(Dist)
        return min(Dist)
```

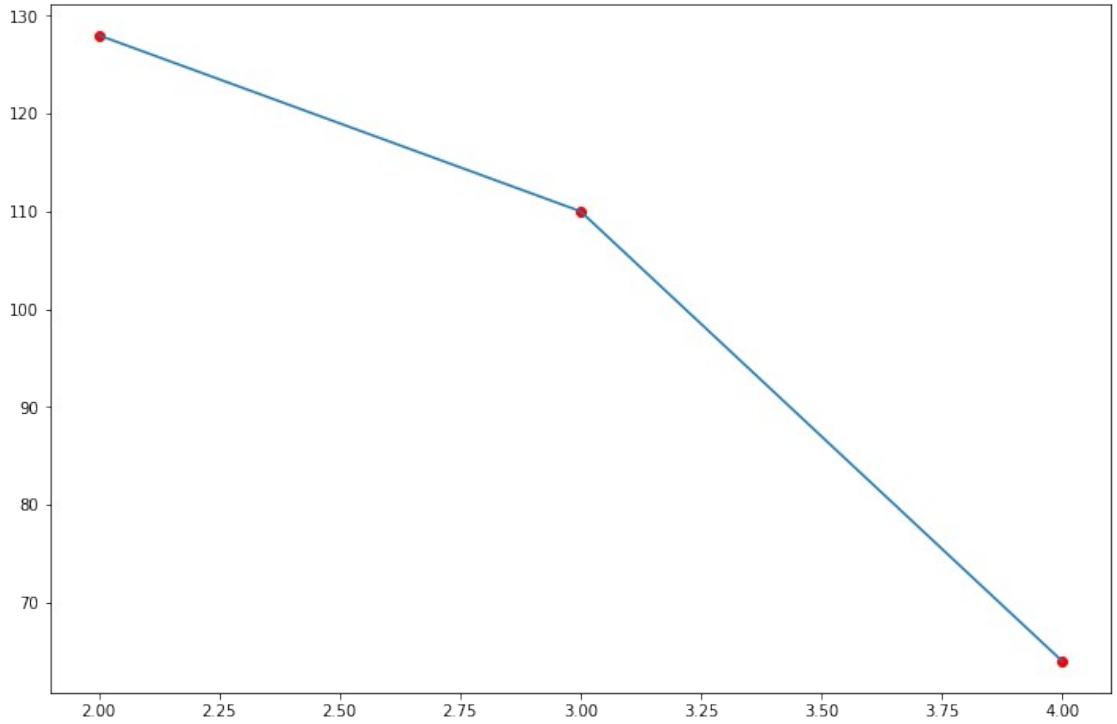
[19]:
```
TSPGetMinDistance(1, 1, [2, 3, 4])
```

[19]: 64

[20]:
```
iterative_process[-1]
```

[20]: [110, 64, 128]

[21]:
```
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 8))
final_costs = sorted(iterative_process[-1])
final_costs = final_costs[::-1]
plt.scatter(list(range(2, len(final_costs) + 2)), final_costs, color='red')
plt.plot(list(range(2, len(final_costs) + 2)), final_costs)
plt.show()
print("časovna razlika :", timeit.default_timer() - starttime)
```




```
časovna razlika : 0.14422274999924412
```

[ ]:

Priloga 6

Problem preusmerjanja vozil klasična in kvantna rešitev

March 5, 2022

[100]:
```
import operator
import sys
import cplex
from cplex.exceptions import CplexError
import numpy as np

import math

from qiskit import BasicAer
from qiskit.quantum_info import Pauli
from qiskit.utils import QuantumInstance, algorithm_globals
from qiskit.algorithms import NumPyMinimumEigensolver, VQE
from qiskit.circuit.library import TwoLocal
from qiskit.algorithms.optimizers import SPSA
import operator
import matplotlib.pyplot as plt
```

[101]:
```
n = 5# določimo število postojank
K = 3 # določimo število vozil
```

[102]:
```
class Initializer:
    def __init__(self, n):
        self.n = n

    def generate_instance(self):

        n = self.n

        np.random.seed(1543)

        xc = (np.random.rand(n) - 0.5) * 10
        yc = (np.random.rand(n) - 0.5) * 10

        instance = np.zeros([n, n])
        for ii in range(0, n):
            for jj in range(ii + 1, n):
```

```
                instance[ii, jj] = (xc[ii] - xc[jj]) ** 2 + (yc[ii] - yc[jj])␣
↪** 2
                instance[jj, ii] = instance[ii, jj]

        return xc, yc, instance
```

[103]:
```
initializer = Initializer(n)
xc, yc, instance = initializer.generate_instance()
```

[104]:
```
class ClassicalOptimizer:
    def __init__(self, instance, n, K):

        self.instance = instance
        self.n = n
        self.K = K

    def compute_allowed_combinations(self):
        f = math.factorial
        return f(self.n) / f(self.K) / f(self.n - self.K)

    def cplex_solution(self):


        instance = self.instance
        n = self.n
        K = self.K

        my_obj = list(instance.reshape(1, n ** 2)[0]) + [0.0 for x in range(0,␣
↪n - 1)]
        my_ub = [1 for x in range(0, n ** 2 + n - 1)]
        my_lb = [0 for x in range(0, n ** 2)] + [0.1 for x in range(0, n - 1)]
        my_ctype = "".join(["I" for x in range(0, n ** 2)]) + "".join(
            ["C" for x in range(0, n - 1)]
        )

        my_rhs = (
            2 * ([K] + [1 for x in range(0, n - 1)])
            + [1 - 0.1 for x in range(0, (n - 1) ** 2 - (n - 1))]
            + [0 for x in range(0, n)]
        )
        my_sense = (
            "".join(["E" for x in range(0, 2 * n)])
            + "".join(["L" for x in range(0, (n - 1) ** 2 - (n - 1))])
            + "".join(["E" for x in range(0, n)])
        )

        try:
```

```
            my_prob = cplex.Cplex()
            self.populatebyrow(my_prob, my_obj, my_ub, my_lb, my_ctype,␣
↪my_sense, my_rhs)

            my_prob.solve()

        except CplexError as exc:
            print(exc)
            return

        x = my_prob.solution.get_values()
        x = np.array(x)
        cost = my_prob.solution.get_objective_value()

        return x, cost

    def populatebyrow(self, prob, my_obj, my_ub, my_lb, my_ctype, my_sense,␣
↪my_rhs):

        n = self.n

        prob.objective.set_sense(prob.objective.sense.minimize)
        prob.variables.add(obj=my_obj, lb=my_lb, ub=my_ub, types=my_ctype)

        prob.set_log_stream(None)
        prob.set_error_stream(None)
        prob.set_warning_stream(None)
        prob.set_results_stream(None)

        rows = []
        for ii in range(0, n):
            col = [x for x in range(0 + n * ii, n + n * ii)]
            coef = [1 for x in range(0, n)]
            rows.append([col, coef])

        for ii in range(0, n):
            col = [x for x in range(0 + ii, n ** 2, n)]
            coef = [1 for x in range(0, n)]

            rows.append([col, coef])

        for ii in range(0, n):
            for jj in range(0, n):
                if (ii != jj) and (ii * jj > 0):

                    col = [ii + (jj * n), n ** 2 + ii - 1, n ** 2 + jj - 1]
                    coef = [1, 1, -1]
```

```
                rows.append([col, coef])

        for ii in range(0, n):
            col = [(ii) * (n + 1)]
            coef = [1]
            rows.append([col, coef])

        prob.linear_constraints.add(lin_expr=rows, senses=my_sense, rhs=my_rhs)
```

[105]:
```
classical_optimizer = ClassicalOptimizer(instance, n, K)

print("Število izvedljivih rešitev = " + str(classical_optimizer.
↪compute_allowed_combinations()))
```

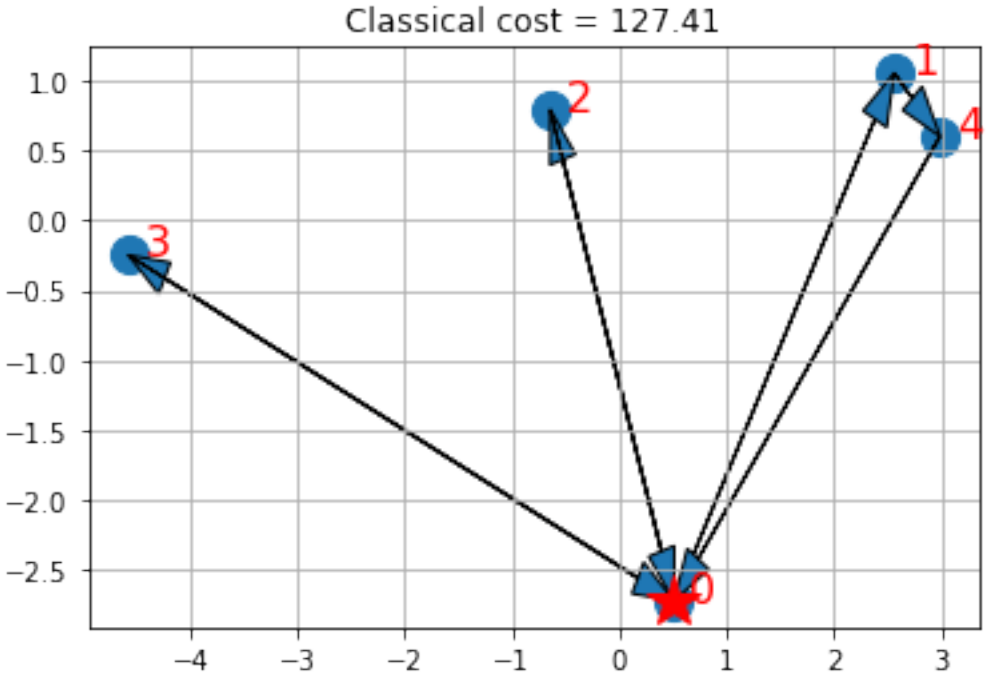
```
Število izvedljivih rešitev = 10.0
```

[106]:
```
x = None
z = None
x, classical_cost = classical_optimizer.cplex_solution()
z = [x[ii] for ii in range(n ** 2) if ii // n != ii % n]
print(z)
```

```
[1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
1.0, 0.0, 0.0, 0.0]
```

[107]:
```
def visualize_solution(xc, yc, x, C, n, K, title_str):
    plt.figure()
    plt.scatter(xc, yc, s=200)
    for i in range(len(xc)):
        plt.annotate(i, (xc[i] + 0.15, yc[i]), size=16, color="r")
    plt.plot(xc[0], yc[0], "r*", ms=20)

    plt.grid()

    for ii in range(0, n ** 2):

        if x[ii] > 0:
            ix = ii // n
            iy = ii % n
            plt.arrow(
                xc[ix],
                yc[ix],
                xc[iy] - xc[ix],
                yc[iy] - yc[ix],
                length_includes_head=True,
                head_width=0.25,
```

```
        )

    plt.title(title_str + " cena = " + str(int(C * 100) / 100.0))
    plt.show()


if x is not None:
    visualize_solution(xc, yc, x, classical_cost, n, K, "Klasična")
```



[108]:
```
from qiskit_optimization import QuadraticProgram
from qiskit_optimization.algorithms import MinimumEigenOptimizer


class QuantumOptimizer:
    def __init__(self, instance, n, K):

        self.instance = instance
        self.n = n
        self.K = K

    def binary_representation(self, x_sol=0):

        instance = self.instance
        n = self.n
```

```
        K = self.K

        A = np.max(instance) * 100  # A parameter of cost function

        instance_vec = instance.reshape(n ** 2)
        w_list = [instance_vec[x] for x in range(n ** 2) if instance_vec[x] > 0]
        w = np.zeros(n * (n - 1))
        for ii in range(len(w_list)):
            w[ii] = w_list[ii]

        Id_n = np.eye(n)
        Im_n_1 = np.ones([n - 1, n - 1])
        Iv_n_1 = np.ones(n)
        Iv_n_1[0] = 0
        Iv_n = np.ones(n - 1)
        neg_Iv_n_1 = np.ones(n) - Iv_n_1

        v = np.zeros([n, n * (n - 1)])
        for ii in range(n):
            count = ii - 1
            for jj in range(n * (n - 1)):

                if jj // (n - 1) == ii:
                    count = ii

                if jj // (n - 1) != ii and jj % (n - 1) == count:
                    v[ii][jj] = 1.0

        vn = np.sum(v[1:], axis=0)

        Q = A * (np.kron(Id_n, Im_n_1) + np.dot(v.T, v))

        g = (
            w
            - 2 * A * (np.kron(Iv_n_1, Iv_n) + vn.T)
            - 2 * A * K * (np.kron(neg_Iv_n_1, Iv_n) + v[0].T)
        )

        c = 2 * A * (n - 1) + 2 * A * (K ** 2)

        try:
            max(x_sol)
            fun = (
                lambda x: np.dot(np.around(x), np.dot(Q, np.around(x)))
                + np.dot(g, np.around(x))
                + c
            )
```

```
            cost = fun(x_sol)
        except:
            cost = 0

        return Q, g, c, cost

    def construct_problem(self, Q, g, c) -> QuadraticProgram:
        qp = QuadraticProgram()
        for i in range(n * (n - 1)):
            qp.binary_var(str(i))
        qp.objective.quadratic = Q
        qp.objective.linear = g
        qp.objective.constant = c
        return qp

    def solve_problem(self, qp):
        algorithm_globals.random_seed = 10598
        quantum_instance = QuantumInstance(
            BasicAer.get_backend("qasm_simulator"),
            seed_simulator=algorithm_globals.random_seed,
            seed_transpiler=algorithm_globals.random_seed,
        )

        vqe = VQE(quantum_instance=quantum_instance)
        optimizer = MinimumEigenOptimizer(min_eigen_solver=vqe)
        result = optimizer.solve(qp)
        _, _, _, level = self.binary_representation(x_sol=result.x)
        return result.x, level
```

[109]:
```
quantum_optimizer = QuantumOptimizer(instance, n, K)
```

[110]:
```
if z is not None:
    Q, g, c, binary_cost = quantum_optimizer.binary_representation(x_sol=z)
    print("Binarna cena:", binary_cost, "klasična cena:", classical_cost)
    else:
        print("Could not verify the correctness, due to CPLEX solution being␣
↪unavailable.")
        Q, g, c, binary_cost = quantum_optimizer.binary_representation()
        print("Binary cost:", binary_cost)
```

```
Binary cost: 127.41191469453042 classical cost: 127.41191469451641
Binary formulation is correct
```

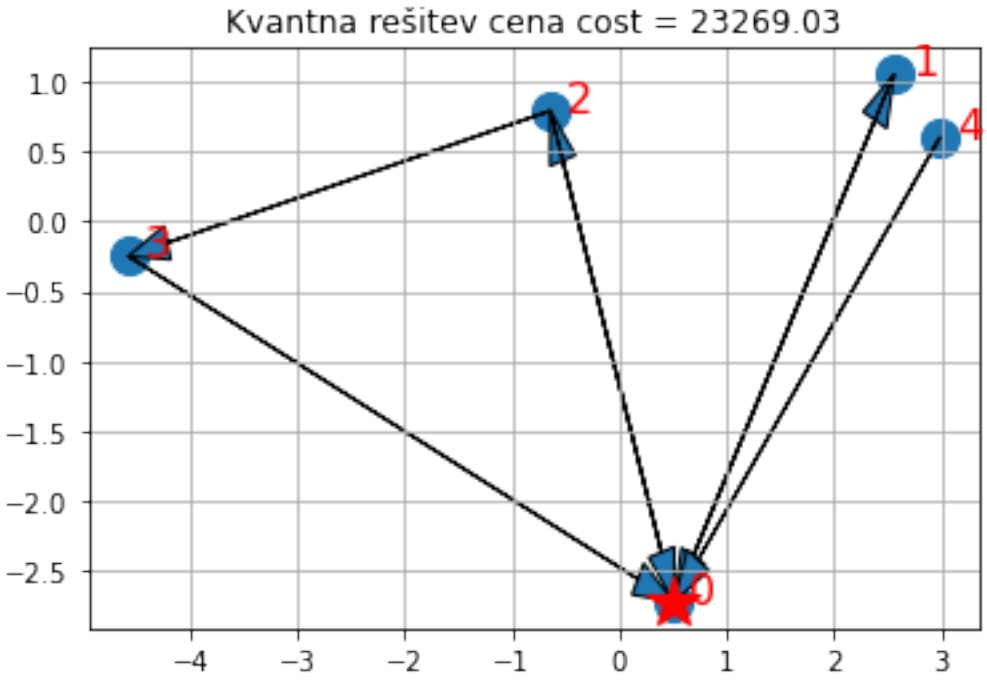
[111]:
```
qp = quantum_optimizer.construct_problem(Q, g, c)
```

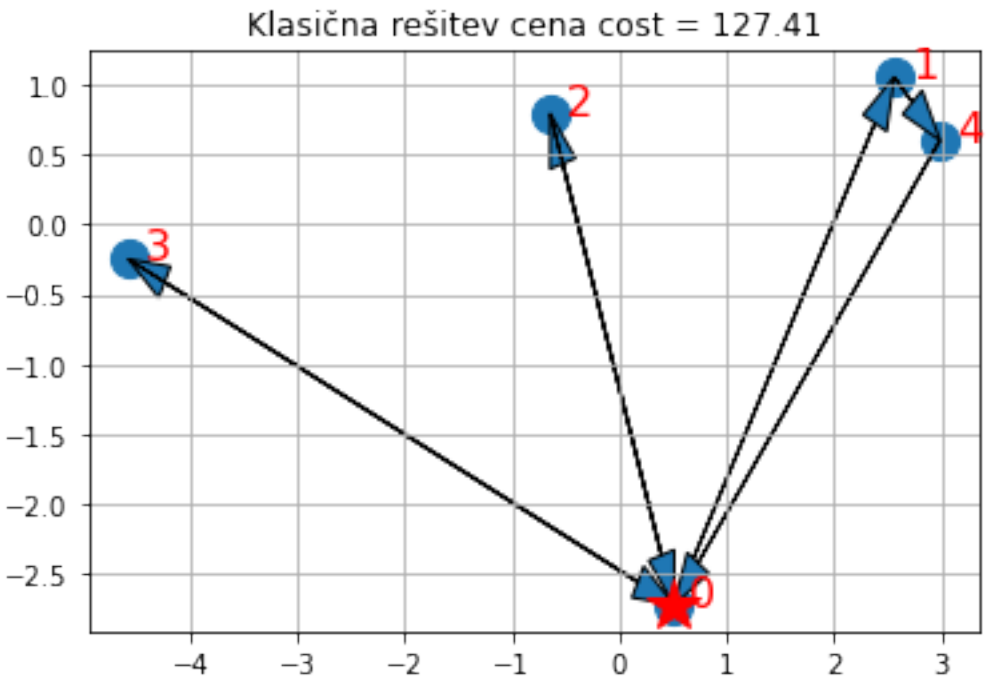
[112]:
```
quantum_solution, quantum_cost = quantum_optimizer.solve_problem(qp)
```

```
print(quantum_solution, quantum_cost)
```

```
[1. 1. 0. 0. 1. 0. 0. 0. 1. 0. 1. 0. 1. 0. 0. 0. 1. 0. 0. 0.] 23269.03850208578
```

[113]:
```
x_quantum = np.zeros(n ** 2)
kk = 0
for ii in range(n ** 2):
    if ii // n != ii % n:
        x_quantum[ii] = quantum_solution[kk]
        kk += 1


visualize_solution(xc, yc, x_quantum, quantum_cost, n, K, "Kvantna rešitev␣
↪cena")

if x is not None:
    visualize_solution(xc, yc, x, classical_cost, n, K, "Klasična rešitev cena")
```





Program izvzet s spletne strani https://qiskit.org/documentation/optimization/tutorials/07_example

Priloga 7

Elelktronska struktura

March 4, 2022

[21]:
```
import matplotlib as plt
from qiskit_nature.drivers import UnitsType, Molecule
from qiskit_nature.drivers.second_quantization import␣
↪ElectronicStructureDriverType, ElectronicStructureMoleculeDriver

molecule = Molecule(
    geometry=[["O", [0.0, 0.0, 0.115]],
              ["H", [0.0, 0.754, -0.459]],
              ["H", [0.0, -0.754, -0.459]]], charge=0, multiplicity=1
)
driver = ElectronicStructureMoleculeDriver(
    molecule=molecule, basis="sto3g", driver_type=ElectronicStructureDriverType.
↪PYSCF
)
```

[22]:
```
from qiskit_nature.problems.second_quantization import␣
↪ElectronicStructureProblem
from qiskit_nature.converters.second_quantization import QubitConverter
from qiskit_nature.mappers.second_quantization import JordanWignerMapper,␣
↪ParityMapper
problem = ElectronicStructureProblem(driver)
second_q_ops = problem.second_q_ops()
hamiltonian = second_q_ops[0]
```

[23]:
```
from qiskit_nature.transformers.second_quantization.electronic import␣
↪ActiveSpaceTransformer
transformer = ActiveSpaceTransformer(
num_electrons=2,
num_molecular_orbitals=3)
```

[24]:
```
problem_reduciran = ElectronicStructureProblem(driver,[transformer])
reduciran_q_ops = problem_reduciran.second_q_ops()
hamiltonian_reduciran = reduciran_q_ops[0]
print(hamiltonian_reduciran)
```

```
Fermionic Operator
register length=6, number terms=33
```

```
  (0.038931030432761145+0j) * ( +_0 -_1 +_3 -_4 )
+ (-0.03893103043276114+0j) * ( +_0 -_1 -_3 +_4 )
+ (0.024354547060212665+0j) * ( +_0 -_2 +_3 -_5 )
+ (-0.024354547060212655+0j) * ( +_0 -_2 -_3 + …
```

[25]:
```
qubit_converter = QubitConverter(mapper=JordanWignerMapper())
qubit_op = qubit_converter.convert(reduciran_q_ops[0])
```

[26]:
```
qubit_converter = QubitConverter(mapper=ParityMapper(),␣
↪two_qubit_reduction=True)
qubit_op = qubit_converter.convert(reduciran_q_ops[0], num_particles=problem.
↪num_particles)
print(qubit_op)
```

```
-0.0513574313237262 * IIII
- 0.6184268772816441 * ZIII
+ (0.28904519744486923-1.3877787807814457e-17j) * IZII
- 0.5764253676977917 * ZZII
+ (0.618426877281644+2.7755575615628914e-17j) * IIZI
- 0.15537730842446273 * ZIZI
+ 0.15665677035437253 * IZZI
+ (-0.14173507851371045+1.3877787807814457e-17j) * ZZZI
- 0.28904519744486945 * IIIZ
+ 0.15665677035437253 * ZIIZ
- 0.22003977334376112 * IZIZ
+ 0.14721911941224639 * ZZIZ
+ (-0.5764253676977916+4.163336342344337e-17j) * IIZZ
+ 0.14173507851371045 * ZIZZ
+ (-0.14721911941224639+1.3877787807814457e-17j) * IZZZ
+ 0.14925817605490505 * ZZZZ
+ 0.028975793459948797 * XIXI
- 0.028975793459948794 * XZXI
+ (0.028975793459948794-1.734723475976807e-18j) * XIXZ
- 0.028975793459948794 * XZXZ
+ 0.00973275760819028 * IXIX
+ (0.00973275760819028+8.673617379884035e-19j) * ZXIX
- 0.00973275760819028 * IXZX
+ (-0.00973275760819028-8.673617379884035e-19j) * ZXZX
+ 0.006088636765053161 * XXXX
- 0.006088636765053161 * YYXX
- 0.006088636765053161 * XXYY
+ 0.006088636765053161 * YYYY
```

[27]:
```
from qiskit.algorithms.optimizers import SLSQP
from qiskit.providers.aer import StatevectorSimulator
from qiskit_nature.algorithms.ground_state_solvers.
↪minimum_eigensolver_factories import VQEUCCFactory
```

```
vqe_factory = VQEUCCFactory(quantum_instance=StatevectorSimulator(),
 ↪optimizer=SLSQP(),
)
```

[28]:
```
from qiskit_nature.algorithms.ground_state_solvers import GroundStateEigensolver
solver = GroundStateEigensolver(qubit_converter, vqe_factory)
```

[29]:
```
result = solver.solve(problem_reduciran)
```

[30]:
```
print(result)
```

```
=== GROUND STATE ENERGY ===

* Electronic ground state energy (Hartree): -84.248285546105
  - computed part:   -1.664149612417
  - ActiveSpaceTransformer extracted energy part: -82.584135933688
~ Nuclear repulsion energy (Hartree): 9.285714221678
> Total ground state energy (Hartree): -74.962571324427

=== MEASURED OBSERVABLES ===

  0:  # Particles: 2.000 S: 0.000 S^2: 0.000 M: 0.000

=== DIPOLE MOMENTS ===

~ Nuclear dipole moment (a.u.): [0.0  0.00377945]

  0:
  * Electronic dipole moment (a.u.): [0.00000001-0.00000001 0.68311968]
    - computed part:   [0.00000001 -0.00000001 0.43252811]
    - ActiveSpaceTransformer extracted energy part: 0.0 [0.0 0.25059156]
  > Dipole moment (a.u.): [-0.00000001 0.00000001 -0.67934023] Total:
0.67934023
                 (debye): [-0.00000002 0.00000002 -1.72671046] Total:
1.72671046
```

Priloga 8

Prelaganje proteinov

March 4, 2022

[1]:
```
#uvozimo potrebne knjižnice in podatke za naš eksperiment
from qiskit_nature.problems.sampling.protein_folding.interactions.
 ↪random_interaction import (
    RandomInteraction,
)
from qiskit_nature.problems.sampling.protein_folding.interactions.
 ↪miyazawa_jernigan_interaction import (
    MiyazawaJerniganInteraction,
)
from qiskit_nature.problems.sampling.protein_folding.peptide.peptide import␣
 ↪Peptide
from qiskit_nature.problems.sampling.protein_folding.protein_folding_problem␣
 ↪import (
    ProteinFoldingProblem,
)

from qiskit_nature.problems.sampling.protein_folding.penalty_parameters import␣
 ↪PenaltyParameters

from qiskit.utils import algorithm_globals, QuantumInstance

algorithm_globals.random_seed = 23
```

[2]:
```
#naredimo glavno verigo proteinov s sedmimi aminokislinami poimenovanimi po␣
 ↪črkah.
glavna_veriga = "APRLRFY"
```

[3]:
```
#in določimo stranske verige
stranske_verige = [""] * 7
```

[4]:
```
#določimo da so interakcije med aminokislinami po modelu od Miyazawe in␣
 ↪Jernigana
naključne_interakcije = RandomInteraction()
mj_interakcije = MiyazawaJerniganInteraction()
```

[5]:
```
#določimo že prej opisane kazni
kazen_nazaj = 10
```

```
kazen_chiral = 10
kazen_1 = 10

penalty_terms = PenaltyParameters(kazen_chiral, kazen_nazaj, kazen_1)
```

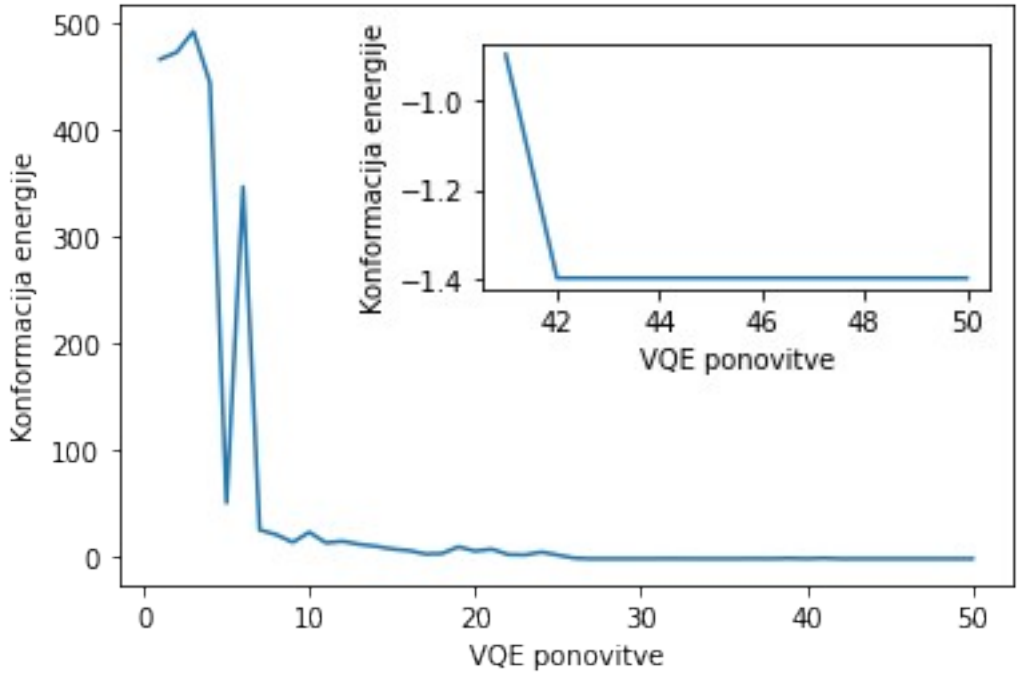
[6]:
```
peptid = Peptide(glavna_veriga, stranske_verige)
```

[7]:
```
prepogibanje_beljakovin = ProteinFoldingProblem(peptid, mj_interakcije,
↪penalty_terms)
qubit_op = prepogibanje_beljakovin.qubit_op()
```

[8]:
```
from qiskit.circuit.library import RealAmplitudes
from qiskit.algorithms.optimizers import COBYLA
from qiskit.algorithms import NumPyMinimumEigensolver, VQE
from qiskit.opflow import PauliExpectation, CVaRExpectation
from qiskit import execute, Aer

#nastavimo klasični optimizator
optimizator = COBYLA(maxiter=50)

#in ansatz
ansatz = RealAmplitudes(reps=1)

#nastavimo simulator, določimo vrsto in število poskusov
backend_simulator = "aer_simulator"
backend = QuantumInstance(
    Aer.get_backend(backend_simulator),
    shots=8192,
    seed_transpiler=algorithm_globals.random_seed,
    seed_simulator=algorithm_globals.random_seed,
)

counts = []
values = []


def shrani_vmesni_rezultat(eval_count, parameters, mean, std):
    counts.append(eval_count)
    values.append(mean)


#uporabimo CVaR_aplha z 0.1
cvar_exp = CVaRExpectation(0.1, PauliExpectation())

# s CVaR zaženemo VQE algoritem
vqe = VQE(
    expectation=cvar_exp,
```

```
    optimizer=optimizator,
    ansatz=ansatz,
    quantum_instance=backend,
    callback=shrani_vmesni_rezultat,
)

result = vqe.compute_minimum_eigenvalue(qubit_op)
```

[9]:
```
#na koncu samo še na grafu prikažemo konformacijo energije v odvisnosti od poskusov algoritma
import matplotlib.pyplot as plt

fig = plt.figure()

plt.plot(counts, values)
plt.ylabel("Konformacija energije")
plt.xlabel("VQE ponovitve")

fig.add_axes([0.44, 0.51, 0.44, 0.32])

plt.plot(counts[40:], values[40:])
plt.ylabel("Konformacija energije")
plt.xlabel("VQE ponovitve")
plt.show()
```



[ ]:

Priloga 9

Kvantna kemija, The Variational Quantum Eigensolver

December 11, 2021

Predstavljajmo si, da želimo simulirati kemijsko rekacijo. Ključno pri reakcijah je spreminjanje energijskih stanj. Reakcije so lahko endotermne ali eksotermne. To pomeni, da moramo izračunati začetno stanje energij in glede na obnašanje energije ugotovimo, kakšna je potekala reakcija. Z vsakim novim delcem se računska zahtevnost eksponentno povečuje. Naš algoritem deluje na osnovi variacijske metode. To pomeni, da s kvantnim računalnikom določimo energijsko stanje sistema s sklepanjem glede na valovno dolžino svetlobnih valov. VQE algoritmom določamo energijo sistema. Valovno dolžino svetlobe spreminjamo, dokler ne pridemo do minimalnega Hamiltoniana (matematični izraz za celotno energijo sistema, Hamiltonian lahko predstavlja energijo molekule, žoge na vzmeti, ...). Je hibriden algoritem, kar pomeni da delno deluje na klasičnem računalniku. Kvantni računalnik izračunava energijo, klasični del pa optimizira varialne parametre. Primer uporabe je izračunavanje energije glede na medatomsko dolžino v molekuli. Kot lahko zdaj tudi pokazali. Najprej rabimo sklep o valovni dolžini. Na ta sklep bodo vplivale geometrija molekule, elektronska ovojnica in število elektronov. Ansatz je izraz za sklep o svetlobnih valovih glede na parametre molekule. To še moramo vkodirati v kvantni računalnik, temu procesu pa rečemo mapping. Nato glede na Ansatz kvantni računalnik naredi merjenja in izračun ter pošlje te informacije v klasični računalnik, ki prilagodi ansatz glede na nove podatke. Kodo ponavljamo, dokler ne dosežemo minimalno energijsko vrednost na tej razdalji, se računalnik premakne na naslednjo razdaljo in tam izračuna energijsko vrednost.

[1]:
```
import numpy as np #za numerične vrednosti
import pylab #za grafično prikazovanje
import copy
from qiskit import BasicAer #simulator za kvantni računalnik
from qiskit.aqua import aqua_globals, QuantumInstance #orodja za delovanje␣
↪eksperimenta
from qiskit.aqua.algorithms import NumPyMinimumEigensolver, VQE #orodje za␣
↪točne energije na osnovi kalkulacije klasičnega računalnika in primerja␣
↪vrednosti našega vqe izračuna
from qiskit.aqua.components.optimizers import SLSQP #klasični optimizator za␣
↪prilagajanje ansatza
from qiskit.chemistry.components.initial_states import HartreeFock #začetni␣
↪ansatz, ki ga bomo sproti prilagajali
from qiskit.chemistry.components.variational_forms import UCCSD #orodje, ki␣
↪pomaga spremeniti začetni ansatz v vqe, spremeni začetno ugibanje v ansatz
from qiskit.chemistry.drivers import PySCFDriver #nastavi/naredi molekulo
from qiskit.chemistry.core import Hamiltonian, QubitMappingType #pomaga␣
↪narediti mapping (vkodira podatke v kvantni računalnik)
```

```
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/aqua/__init__.py:86: DeprecationWarning: The package qiskit.aqua
is deprecated. It was moved/refactored to qiskit-terra For more information see
<https://github.com/Qiskit/qiskit-aqua/blob/main/README.md#migration-guide>
  warn_package('aqua', 'qiskit-terra')
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/chemistry/__init__.py:170: DeprecationWarning: The package
qiskit.chemistry is deprecated. It was moved/refactored to qiskit_nature (pip
install qiskit-nature). For more information see
<https://github.com/Qiskit/qiskit-aqua/blob/main/README.md#migration-guide>
  warn_package('chemistry', 'qiskit_nature', 'qiskit-nature')
```

[2]:
```
molekula = 'H .0 .0 -{0}; Li .0 .0 {0}'
#nastavimo molekule v 3 dimenzionalen prostor. Naredimo molekulo LiH. Dodoamo␣
↪razdalje in določimo na kateri osi se dogajajo spremembe {}. V našem primeru␣
↪na z osi
dolžina = np.arange(0.5,4.25,0.25)
#naredimo arraye na katerih dolžinah naj računalnik meri. Od 0,5 do 4,25␣
↪angstromih v intervalih po 0,25
VQE_energije = []
#tu bomo zapisali energijsko stanje, ki ga bo računalnik dobil
hf_energije = []
#zapiše ugibano energijsko stanje pred optimizacijo algoritma. Tega smo že␣
↪določili zgoraj med importi.
dejanska_energija = []
#zapiše dejansko energijo, da primerjamo svoje rezultate z njo
```

[3]:
```
for i,d in enumerate(dolžina):
    print('korak',i)
#naredimo loop kjer gremo čez različne razdalje in računamo vqe. Vmes še nam␣
↪pokaže v katerem koraku računanje je.
    driver = PySCFDriver(molekula.format(d/2), basis='sto3g')
    #nastavimo driver ki bo nastavil eksperiment za nas. Nato dodamo noter našo␣
↪molekulo,
    #nastavimo spreminjanje razdalje in računanje energije na vsaki in␣
↪nastavimo basis ki bo določalo kako driver določa elektronsko ovojnico.␣
↪Izbral sem sto3g
    qmolekula = driver.run()
    #nastavimo molekulo za klasično simulacijo
    operator = Hamiltonian(qubit_mapping=QubitMappingType.PARITY, #dodamo␣
↪hamiltonian in mapping. Uporabim način parity
                           two_qubit_reduction=True, freeze_core=True,
    #naredimo trik za pospeševanje reakcije two qubit reduction in zamrznemo␣
↪jedro ki nima vpliva na vezave, za hitrejšo reakcijo.
                           orbital_reduction=[-3,-2])
    #zmanjašamo orbitali ki ne vplivata na naše kalkulacije.
    qubit_op, aux_ops = operator.run(qmolekula)
```

```
   #uporabimo operator za klasične operacije
   dejanski_rezultat = NumPyMinimumEigensolver(qubit_op,␣
↪aux_operators=aux_ops).run()
   dejanski_rezultat = operator.process_algorithm_result(dejanski_rezultat)
   #dejanski klasični rezultati za primerjavo z VQEjem

   #VQE:
   optimizator = SLSQP(maxiter=1000)
   #definiramo optimizator in nastavimo max število poskusov na 1000
   začetno_stanje = HartreeFock(operator.molecule_info['num_orbitals'],
                                operator.molecule_info['num_particles'],␣
↪#določimo začetno sklepanje valovne dolžine in vnesemo število orbital in␣
↪delcev
                                qubit_mapping=operator._qubit_mapping,␣
↪#dolločimo še mapping
                                two_qubit_reduction=operator.
↪_two_qubit_reduction) #in two qubit redcution ki bo pospešil našo kalkulacijo
   var_form = UCCSD(num_orbitals=operator.molecule_info['num_orbitals'],
                    num_particles=operator.molecule_info['num_particles'],
                    initial_state = začetno_stanje,
                     qubit_mapping=operator._qubit_mapping,
                    two_qubit_reduction=operator._two_qubit_reduction)
   #v tem delu vzamemo začetno stanje in delamo variacije glede na merjenja.␣
↪Določimo potrebne informacije
   algo = VQE(qubit_op, var_form, optimizator, aux_operators=aux_ops)

   vqe_result = algo.run(QuantumInstance(BasicAer.
↪get_backend('statevector_simulator')))
   vqe_result = operator.process_algorithm_result(vqe_result)

   dejanska_energija.append(dejanski_rezultat.energy)
   VQE_energije.append(vqe_result.energy)
   hf_energije.append(vqe_result.hartree_fock_energy)
```

```
korak 0
```

```
/tmp/ipykernel_157/189934931.py:9: DeprecationWarning: The Hamiltonian class is
deprecated as of Qiskit Aqua 0.8.0 and will be removed no earlier than 3 months
after the release date. Instead, the FermionicTransformation can be used.
  operator = Hamiltonian(qubit_mapping=QubitMappingType.PARITY, #dodamo
hamiltonian in mapping. Uporabim način parity
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/chemistry/core/hamiltonian.py:88: DeprecationWarning: The
ChemistryOperator is deprecated as of Qiskit Aqua 0.8.0 and will be removed no
earlier than 3 months after the release date. Instead, the
FermionicTransformation can be used to transform QMolecules and construct ground
state result objects.
  super().__init__()
```

```
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/chemistry/fermionic_operator.py:386: DeprecationWarning: The
package qiskit.aqua.operators is deprecated. It was moved/refactored to
qiskit.opflow (pip install qiskit-terra). For more information see
<https://github.com/Qiskit/qiskit-aqua/blob/main/README.md#migration-guide>
  pauli_list = WeightedPauliOperator(paulis=[])
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/chemistry/fermionic_operator.py:394: DeprecationWarning: The
variable qiskit.aqua.aqua_globals is deprecated. It was moved/refactored to
qiskit.utils.algorithm_globals (pip install qiskit-terra). For more information
see <https://github.com/Qiskit/qiskit-aqua/blob/main/README.md#migration-guide>
  task_args=(threshold,), num_processes=aqua_globals.num_processes)
/home/coolmodel/miniconda3/lib/python3.9/site-packages/qiskit/aqua/algorithms/mi
nimum_eigen_solvers/minimum_eigen_solver.py:36: DeprecationWarning: The package
qiskit.aqua.algorithms.minimum_eigen_solvers is deprecated. It was
moved/refactored to qiskit.algorithms.minimum_eigen_solvers (pip install qiskit-
terra). For more information see <https://github.com/Qiskit/qiskit-
aqua/blob/main/README.md#migration-guide>
  warn_package('aqua.algorithms.minimum_eigen_solvers',
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/aqua/algorithms/eigen_solvers/eigen_solver.py:36:
DeprecationWarning: The package qiskit.aqua.algorithms.eigen_solvers is
deprecated. It was moved/refactored to qiskit.algorithms.eigen_solvers (pip
install qiskit-terra). For more information see
<https://github.com/Qiskit/qiskit-aqua/blob/main/README.md#migration-guide>
  warn_package('aqua.algorithms.eigen_solvers',
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/chemistry/core/chemistry_operator.py:170: DeprecationWarning:
The qiskit.chemistry.chemistry_operator.MolecularChemistryResult object is
deprecated as of 0.8.0 and will be removed no sooner than 3 months after the
release. You should use
qiskit.chemistry.algorithms.ground_state_solvers.FermionicGroundStateResult
instead.
  super().__init__(a_dict)
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/chemistry/core/hamiltonian.py:380: DeprecationWarning: The
qiskit.chemistry.chemistry_operator.MolecularGroundStateResult object is
deprecated as of 0.8.0 and will be removed no sooner than 3 months after the
release. You should use
qiskit.chemistry.algorithms.ground_state_solvers.FermionicGroundStateResult
instead.
  mgsr = MolecularGroundStateResult()
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/aqua/components/optimizers/optimizer.py:49: DeprecationWarning:
The package qiskit.aqua.components.optimizers is deprecated. It was
moved/refactored to qiskit.algorithms.optimizers (pip install qiskit-terra). For
more information see <https://github.com/Qiskit/qiskit-
aqua/blob/main/README.md#migration-guide>
```

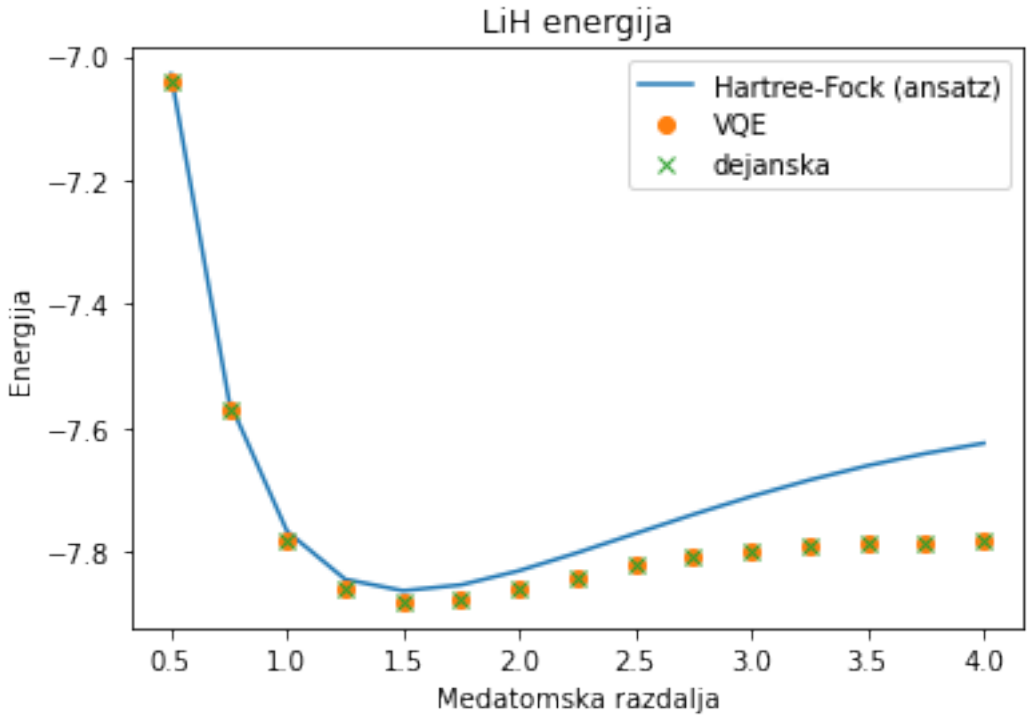
```
  warn_package('aqua.components.optimizers',
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/chemistry/components/initial_states/hartree_fock.py:61:
DeprecationWarning: The HartreeFock class is deprecated as of Aqua 0.9 and will
be removed no earlier than 3 months after the release date. Instead, all
algorithms and circuits accept a plain QuantumCircuit.
  super().__init__()
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/aqua/components/variational_forms/variational_form.py:48:
DeprecationWarning: The package qiskit.aqua.components.variational_forms is
deprecated. For more information see <https://github.com/Qiskit/qiskit-
aqua/blob/main/README.md#migration-guide>
  warn_package('aqua.components.variational_forms')
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/aqua/algorithms/vq_algorithm.py:70: DeprecationWarning: The
class qiskit.aqua.algorithms.VQAlgorithm is deprecated. It was moved/refactored
to qiskit.algorithms.VariationalAlgorithm (pip install qiskit-terra). For more
information see <https://github.com/Qiskit/qiskit-
aqua/blob/main/README.md#migration-guide>
  warn_class('aqua.algorithms.VQAlgorithm',
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/aqua/quantum_instance.py:135: DeprecationWarning: The class
qiskit.aqua.QuantumInstance is deprecated. It was moved/refactored to
qiskit.utils.QuantumInstance (pip install qiskit-terra). For more information
see <https://github.com/Qiskit/qiskit-aqua/blob/main/README.md#migration-guide>
  warn_class('aqua.QuantumInstance',
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/chemistry/components/variational_forms/uccsd.py:429:
DeprecationWarning: Back-references to from Bit instances to their containing
Registers have been deprecated. Instead, inspect Registers to find their
contained Bits.
  qbits[i] = circuit.qubits[qbits[i].index]
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/chemistry/components/variational_forms/uccsd.py:429:
DeprecationWarning: Back-references to from Bit instances to their containing
Registers have been deprecated. Instead, inspect Registers to find their
contained Bits.
  qbits[i] = circuit.qubits[qbits[i].index]
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/chemistry/core/chemistry_operator.py:170: DeprecationWarning:
The qiskit.chemistry.chemistry_operator.MolecularChemistryResult object is
deprecated as of 0.8.0 and will be removed no sooner than 3 months after the
release. You should use
qiskit.chemistry.algorithms.ground_state_solvers.FermionicGroundStateResult
instead.
  super().__init__(a_dict)
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/chemistry/core/hamiltonian.py:380: DeprecationWarning: The
```

```
qiskit.chemistry.chemistry_operator.MolecularGroundStateResult object is
deprecated as of 0.8.0 and will be removed no sooner than 3 months after the
release. You should use
qiskit.chemistry.algorithms.ground_state_solvers.FermionicGroundStateResult
instead.
  mgsr = MolecularGroundStateResult()
/tmp/ipykernel_157/189934931.py:9: DeprecationWarning: The Hamiltonian class is
deprecated as of Qiskit Aqua 0.8.0 and will be removed no earlier than 3 months
after the release date. Instead, the FermionicTransformation can be used.
  operator = Hamiltonian(qubit_mapping=QubitMappingType.PARITY, #dodamo
hamiltonian in mapping. Uporabim način parity
/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/chemistry/core/hamiltonian.py:88: DeprecationWarning: The
ChemistryOperator is deprecated as of Qiskit Aqua 0.8.0 and will be removed no
earlier than 3 months after the release date. Instead, the
FermionicTransformation can be used to transform QMolecules and construct ground
state result objects.
  super().__init__()

korak 1

/home/coolmodel/miniconda3/lib/python3.9/site-
packages/qiskit/chemistry/components/initial_states/hartree_fock.py:61:
DeprecationWarning: The HartreeFock class is deprecated as of Aqua 0.9 and will
be removed no earlier than 3 months after the release date. Instead, all
algorithms and circuits accept a plain QuantumCircuit.
  super().__init__()

korak 2
korak 3
korak 4
korak 5
korak 6
korak 7
korak 8
korak 9
korak 10
korak 11
korak 12
korak 13
korak 14
```

[4]:
```
pylab.plot(dolžina,hf_energije, label='Hartree-Fock (ansatz)')
pylab.plot(dolžina,VQE_energije,'o',label='VQE')
pylab.plot(dolžina,dejanska_energija,'x',label='dejanska')

pylab.xlabel('Medatomska razdalja')
pylab.ylabel('Energija')
pylab.title('LiH energija')
```

```
pylab.legend(loc='upper right')
```

[4]: <matplotlib.legend.Legend at 0x7f5b78103a90>



narišemo graf razmerja med meadatomsko razdaljo in osnovno energijsko vrednostjo. Vidimo, da je najprej VQE sledil Harttree-Focku (našemu ansatzu), pozneje pa je pravilno prilagodil ugibanje, da je bilo enako dejanskemu stanju energije glede na medatomsko razdaljo.

Priloga 10

Kernel kvantno strojno učenje

March 5, 2022

Delamo kvantni support vector machine. Klasični support vector machine je sposoben klasificirati podatke v dve različni skupini. Gre za nadzorovano strojno učenje. Za boljšo razlago si predstavljajmo akvarij z dvema različnima vrstama, ki se v nekem času (hranjenja) razporedijo na del akvarija značilen za le to vrsto. Tako lahko potegnemo črto oziroma hiperravnino, ki nam glede na lokacijo osebka pove kakšne vrste je opazovana žival. Pri hranjenju, se ena vrsta zadržuje skupaj, medtem ko druga pa se naključno razporedi po akvariju. Kako ločiti vrsti sedaj? Predstavimo mapo značilnosti, tako da bomo locirali vsako posamezno žival in alokacijo poslali iz 2d okolja v višjo dimenzijo k. Zdaj lahko izračunamo optimalno hiperravnino v višji dimenziji, ki loči vrsti med seboj. To pomeni da moramo računati razdalje v višji dimenziji, kar bi v velikem k pomenilo veliko računalniško moč. Uporabimo lahko Kernel trick. To je enostavna funkcija, ki vzame dve točki podatka in izračuna razdaljo. Nekatere kernel matrice so težko preračunljive klasično, zato za to uporabljamo kvantne, ki uporabljajo multidimenzionalen prostor računanja za iskanje hiperravnine. Tako se mapa prestavi v višjo dimenzijo, kjer je včasih samo mogoče narediti razliko.

[1]:
```
import qiskit
import cvxpy
```

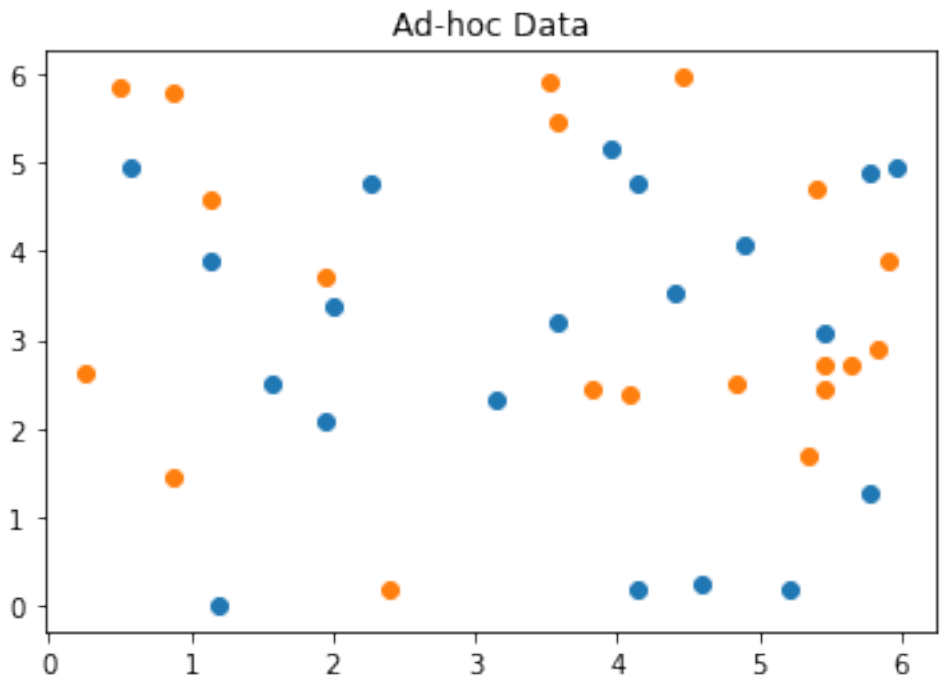
[2]:
```
from matplotlib import pyplot as plt #za grafe in vizualizacijo
import numpy as np #za števila
from qiskit.ml.datasets import ad_hoc_data #nabor podatkov. Lahko uporabljamo␣
↪različne
from qiskit import BasicAer #simulator
from qiskit.aqua import QuantumInstance #za zagon eksperimenta
from qiskit.circuit.library import ZZFeatureMap #mapa značilnosti
from qiskit.aqua.algorithms import QSVM #kvantni statevector (glavna funkcija)
from qiskit.aqua.utils import split_dataset_to_data_and_labels,␣
↪map_label_to_class_name
```

```
C:\Users\aleks\anaconda3\lib\site-packages\qiskit\aqua\__init__.py:86:
DeprecationWarning: The package qiskit.aqua is deprecated. It was
moved/refactored to qiskit-terra For more information see
<https://github.com/Qiskit/qiskit-aqua/blob/main/README.md#migration-guide>
  warn_package('aqua', 'qiskit-terra')
C:\Users\aleks\anaconda3\lib\site-packages\qiskit\ml\__init__.py:40:
DeprecationWarning: The package qiskit.ml is deprecated. It was moved/refactored
to qiskit_machine_learning (pip install qiskit-machine-learning). For more
information see <https://github.com/Qiskit/qiskit-
aqua/blob/main/README.md#migration-guide>
```

```
  warn_package('ml', 'qiskit_machine_learning', 'qiskit-machine-learning')
```

[3]:
```
začetna_dimenzija = 2
velikost_baze_podatkov_za_trening = 20
velikost_baze_podatkov_za_testiranje = 10
random_seed = 10598
shots = 10000

#povemo od kod naj vzame podatke kot so podatkovna baza za treniranje in␣
↪testiranje, ...
sample_Total, training_input, test_input, class_labels =␣
↪ad_hoc_data(training_size=velikost_baze_podatkov_za_trening,

                                                              ␣
↪test_size=velikost_baze_podatkov_za_testiranje,

                                                               gap=0.3,␣
↪#presledek v višji dimenziji, ki bo ločil podatke

                                                              ␣
↪n=začetna_dimenzija,

                                                              ␣
↪plot_data=True)
datapoints, class_to_label = split_dataset_to_data_and_labels(test_input)
print(class_to_label)
```

```
C:\Users\aleks\anaconda3\lib\site-packages\qiskit\ml\datasets\ad_hoc.py:79:
DeprecationWarning: The variable qiskit.aqua.aqua_globals is deprecated. It was
moved/refactored to qiskit.utils.algorithm_globals (pip install qiskit-terra).
For more information see <https://github.com/Qiskit/qiskit-
aqua/blob/main/README.md#migration-guide>
  basis = aqua_globals.random.random((2 ** n, 2 ** n)) + \
```




```
{'A': 0, 'B': 1}
```

vidimo podatke za treniranje računalnika, da bo pozneje znal kvalificirati testne podatke

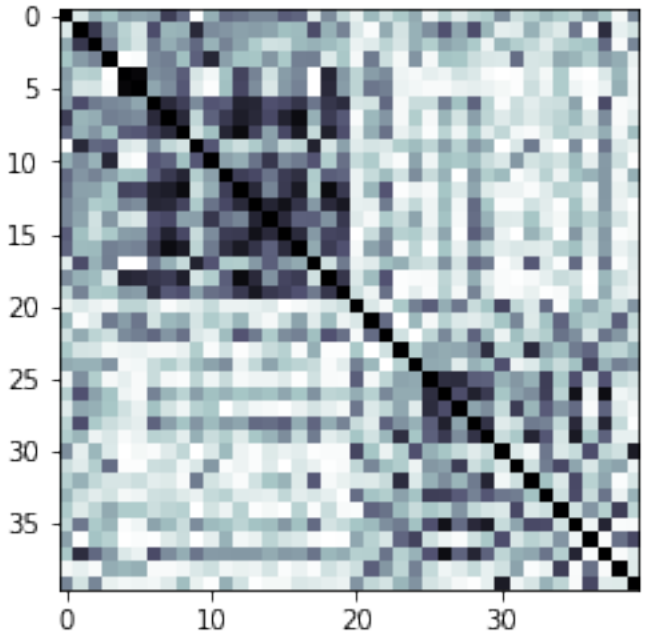
[4]:
```
backend = BasicAer.get_backend('qasm_simulator') #nastavimo simulator
feature_map = ZZFeatureMap(začetna_dimenzija, reps=2) #nastavimo v naprej
↪določeno značilnostno mapo
svm = QSVM(feature_map, training_input, test_input, None)#nastavimo statevector
↪machine: noter damo mapo, podatke za treniranje in testiranje
svm.random_seed = random_seed #najdemo naključno vrednost
eksperiment = QuantumInstance(backend, shots=shots, seed_simulator=random_seed,
↪seed_transpiler=random_seed) #zaženemo eksperiment s quantum instance
rezultati = svm.run(eksperiment)
```

```
C:\Users\aleks\anaconda3\lib\site-
packages\qiskit\aqua\algorithms\classifiers\qsvm\qsvm.py:102:
DeprecationWarning: The package qiskit.aqua.algorithms.classifiers is
deprecated. It was moved/refactored to
qiskit_machine_learning.algorithms.classifiers (pip install qiskit-machine-
learning). For more information see <https://github.com/Qiskit/qiskit-
aqua/blob/main/README.md#migration-guide>
  warn_package('aqua.algorithms.classifiers',
C:\Users\aleks\anaconda3\lib\site-packages\qiskit\aqua\quantum_instance.py:135:
DeprecationWarning: The class qiskit.aqua.QuantumInstance is deprecated. It was
moved/refactored to qiskit.utils.QuantumInstance (pip install qiskit-terra). For
```

```
more information see <https://github.com/Qiskit/qiskit-
aqua/blob/main/README.md#migration-guide>
  warn_class('aqua.QuantumInstance',
```

[6]:
```
print("kernel matrica med treningom:") #izrišemo kako izgleda matrica med␣
↪treningom
kernel_matrix = rezultati['kernel_matrix_training']
slika = plt.imshow(np.
↪asmatrix(kernel_matrix),interpolation='nearest',origin='upper',cmap='bone_r')
```

```
kernel matrica med treningom:
```



Vidimo črne kvadratke na diagonali. To pomeni, da je razdalja od vsake točke do same sebe 0. Tu vidimo izračunano razdaljo med kernelom v višjem dimenzijskem prostoru. Poglejmo še kako dobro je naš qsvm uganil podatke:

[8]:
```
#tukaj je napisano kako je naš algoritem ugibal, kakšni so resnični podatki in␣
↪kakšno je razmerje med ugibanjem in resničnimi podatki
predicted_labels = svm.predict(datapoints[0])
predicted_classes = map_label_to_class_name(predicted_labels, svm.
↪label_to_class)
print("zares: {}".format(datapoints[1]))
print("napoved: {}".format(predicted_labels))
print("testiranje razmerje: ", rezultati['testing_accuracy'])
```

```
zares:  [0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1]
```

```
napoved: [0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1]
testiranje razmerje1.0
```

[ ]:

Priloga 11

kernel machine learning

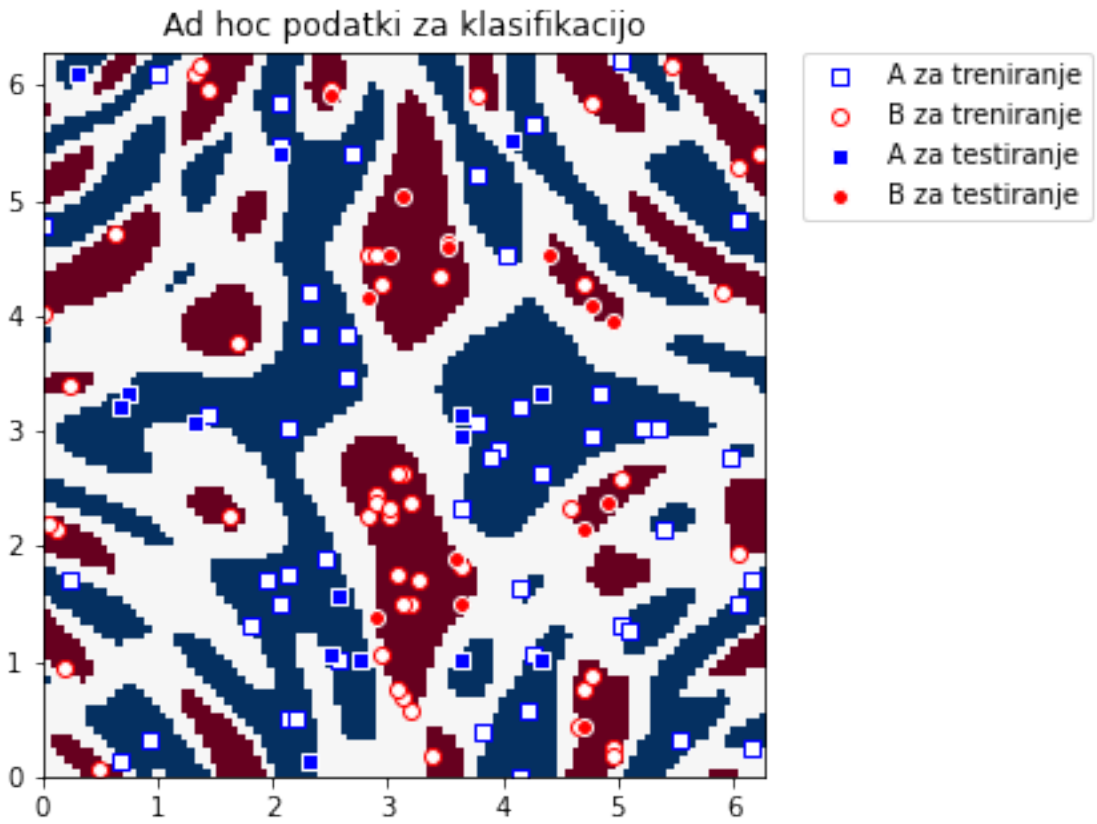
March 6, 2022

[11]:
```
# uvozimo vse potrebne knjižnice in pakete
import matplotlib.pyplot as plt
import numpy as np

from sklearn.svm import SVC
from sklearn.cluster import SpectralClustering
from sklearn.metrics import normalized_mutual_info_score

from qiskit import BasicAer
from qiskit.circuit.library import ZZFeatureMap
from qiskit.utils import QuantumInstance, algorithm_globals
from qiskit_machine_learning.algorithms import QSVC
from qiskit_machine_learning.kernels import QuantumKernel
from qiskit_machine_learning.datasets import ad_hoc_data
import timeit
seed = 12345
algorithm_globals.random_seed = seed
```

[12]:
```
# definiramo podatkovno bazo ad_hoc
adhoc_dimension = 2
train_features, train_labels, test_features, test_labels, adhoc_total =
 ↪ad_hoc_data(
    training_size=50,
    test_size=15,
    n=adhoc_dimension,
    gap=0.3,
    plot_data=False,
    one_hot=False,
    include_sample_total=True,
)
```

[13]:
```
# na gafu prikažemo podatkovno bazo. Prikažemo podatke za testiranje in
 ↪treniranje
plt.figure(figsize=(5, 5))
plt.ylim(0, 2 * np.pi)
plt.xlim(0, 2 * np.pi)
plt.imshow(
```

```
    np.asmatrix(adhoc_total).T,
    interpolation="nearest",
    origin="lower",
    cmap="RdBu",
    extent=[0, 2 * np.pi, 0, 2 * np.pi],
)

plt.scatter(
    train_features[np.where(train_labels[:] == 0), 0],
    train_features[np.where(train_labels[:] == 0), 1],
    marker="s",
    facecolors="w",
    edgecolors="b",
    label="A za treniranje",
)
plt.scatter(
    train_features[np.where(train_labels[:] == 1), 0],
    train_features[np.where(train_labels[:] == 1), 1],
    marker="o",
    facecolors="w",
    edgecolors="r",
    label="B za treniranje",
)
plt.scatter(
    test_features[np.where(test_labels[:] == 0), 0],
    test_features[np.where(test_labels[:] == 0), 1],
    marker="s",
    facecolors="b",
    edgecolors="w",
    label="A za testiranje",
)
plt.scatter(
    test_features[np.where(test_labels[:] == 1), 0],
    test_features[np.where(test_labels[:] == 1), 1],
    marker="o",
    facecolors="r",
    edgecolors="w",
    label="B za testiranje",
)

plt.legend(bbox_to_anchor=(1.05, 1), loc="upper left", borderaxespad=0.0)
plt.title("Ad hoc podatki za klasifikacijo")

plt.show()
```



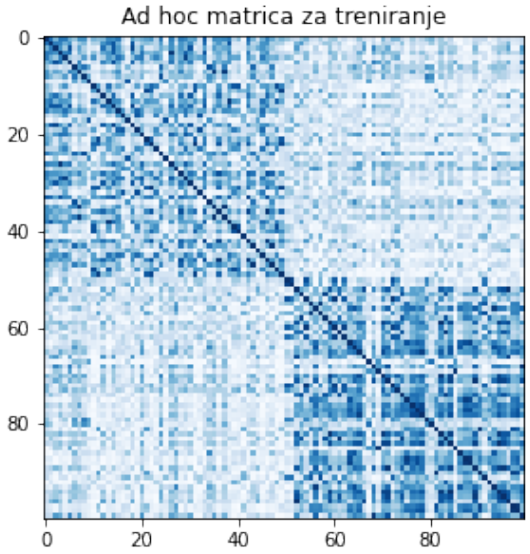
[14]:
```
# določimo mapo lastnosti (uporabljamo ZZFeature) in simulator, zaženemo␣
↪algoritem
adhoc_feature_map = ZZFeatureMap(feature_dimension=adhoc_dimension, reps=2,␣
↪entanglement="linear")

adhoc_backend = QuantumInstance(
    BasicAer.get_backend("qasm_simulator"), shots=1024, seed_simulator=seed,␣
↪seed_transpiler=seed
)

adhoc_kernel = QuantumKernel(feature_map=adhoc_feature_map,␣
↪quantum_instance=adhoc_backend)
```

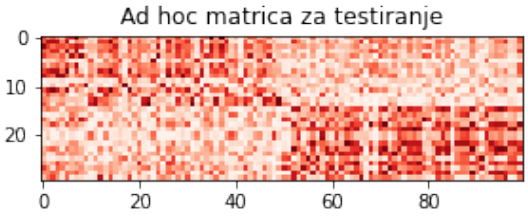
[15]:
```
# preizkusimo klasifikacijo s klasičnim SVC algoritmom
starttime = timeit.default_timer()
adhoc_svc = SVC(kernel=adhoc_kernel.evaluate)
adhoc_svc.fit(train_features, train_labels)
adhoc_score = adhoc_svc.score(test_features, test_labels)

print(f"Rezultat Kernel klasifikacije: {adhoc_score}")
```

```
print("čas:", timeit.default_timer()-starttime)
```

```
Rezultat Kernel klasifikacije: 1.0
čas: 23.197948047999944
```

[16]:
```
# določimo matrico treniranja s katero bomo klasificirali testne podatke.
Matrici prikažemo
adhoc_matrix_train = adhoc_kernel.evaluate(x_vec=train_features)
adhoc_matrix_test = adhoc_kernel.evaluate(x_vec=test_features,
y_vec=train_features)

fig, axs = plt.subplots(1, 2, figsize=(10, 5))
axs[0].imshow(
    np.asmatrix(adhoc_matrix_train), interpolation="nearest", origin="upper",
cmap="Blues"
)
axs[0].set_title("Ad hoc matrica za treniranje")
axs[1].imshow(np.asmatrix(adhoc_matrix_test), interpolation="nearest",
origin="upper", cmap="Reds")
axs[1].set_title("Ad hoc matrica za testiranje")
plt.show()

adhoc_svc = SVC(kernel="precomputed")
adhoc_svc.fit(adhoc_matrix_train, train_labels)
adhoc_score = adhoc_svc.score(adhoc_matrix_test, test_labels)

print(f"Rezultat klasifikacije testiranja: {adhoc_score}")
```





```
Rezultat klasifikacije testiranja: 1.0
```

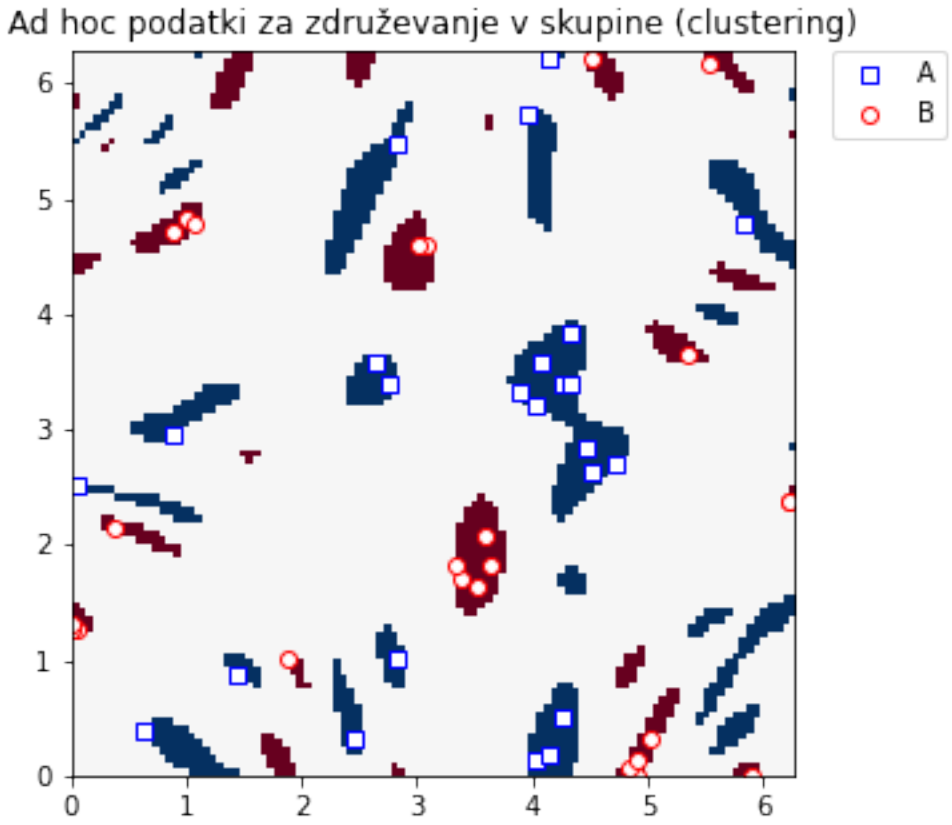
[17]:
```
# določimo kvantni algoritem QSVC in z njim izvedemo model
starttime2 = timeit.default_timer()
qsvc = QSVC(quantum_kernel=adhoc_kernel)
qsvc.fit(train_features, train_labels)
qsvc_score = qsvc.score(test_features, test_labels)

print(f"rezultat klasifikacije testiranja s QSVC algoritmom: {qsvc_score}")
print("čas:", timeit.default_timer()-starttime2)
```

```
rezultat klasifikacije testiranja s QSVC algoritmom: 1.0
čas: 23.101598319000004
```

Združevanje v skupine (clustering)

[18]:
```
# tako kot prej definiramo podatkovno bazo in prikažemo to na grafu
adhoc_dimension = 2
train_features, train_labels, test_features, test_labels, adhoc_total =
 ↪ad_hoc_data(
    training_size=25,
    test_size=0,
    n=adhoc_dimension,
    gap=0.6,
    plot_data=False,
    one_hot=False,
    include_sample_total=True,
)

plt.figure(figsize=(5, 5))
plt.ylim(0, 2 * np.pi)
plt.xlim(0, 2 * np.pi)
plt.imshow(
    np.asmatrix(adhoc_total).T,
    interpolation="nearest",
    origin="lower",
    cmap="RdBu",
    extent=[0, 2 * np.pi, 0, 2 * np.pi],
)
plt.scatter(
    train_features[np.where(train_labels[:] == 0), 0],
    train_features[np.where(train_labels[:] == 0), 1],
    marker="s",
    facecolors="w",
    edgecolors="b",
    label="A",
)
plt.scatter(
```

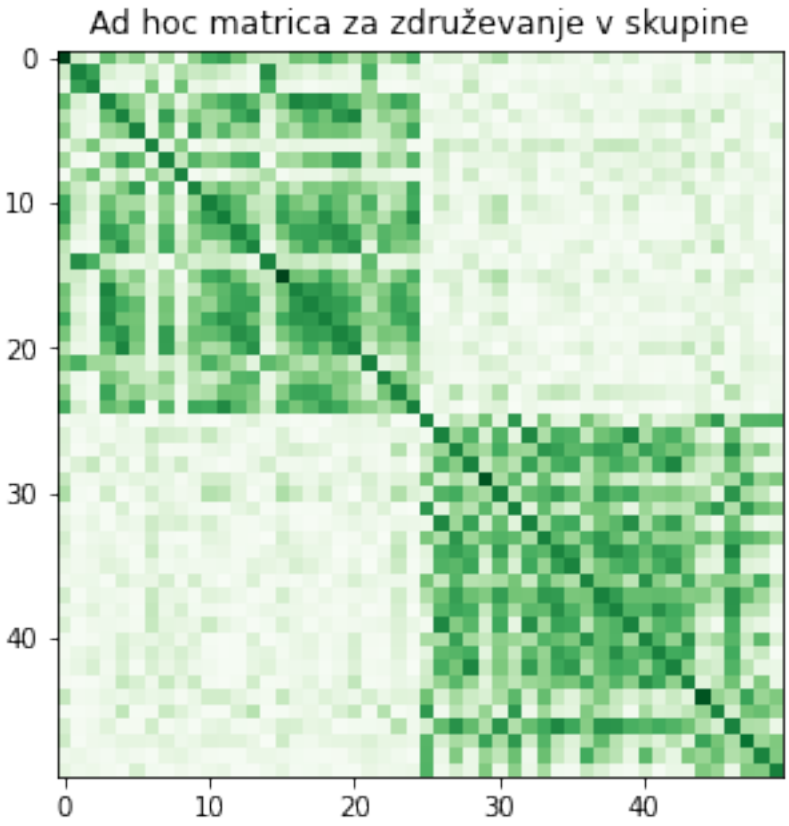
```
    train_features[np.where(train_labels[:] == 1), 0],
    train_features[np.where(train_labels[:] == 1), 1],
    marker="o",
    facecolors="w",
    edgecolors="r",
    label="B",
)

plt.legend(bbox_to_anchor=(1.05, 1), loc="upper left", borderaxespad=0.0)
plt.title("Ad hoc podatki za združevanje v skupine (clustering)")

plt.show()
```



[19]:
```
# uporabljamo isto mapo, simulator in algoritem
adhoc_feature_map = ZZFeatureMap(feature_dimension=adhoc_dimension, reps=2,␣
↪entanglement="linear")

adhoc_backend = QuantumInstance(
    BasicAer.get_backend("qasm_simulator"), shots=1024, seed_simulator=seed,␣
↪seed_transpiler=seed
```

```
)

adhoc_kernel = QuantumKernel(feature_map=adhoc_feature_map,␣
↪quantum_instance=adhoc_backend)
```

[20]:
```
adhoc_matrix = adhoc_kernel.evaluate(x_vec=train_features)

plt.figure(figsize=(5, 5))
plt.imshow(np.asmatrix(adhoc_matrix), interpolation="nearest", origin="upper",␣
↪cmap="Greens")
plt.title("Ad hoc matrica za združevanje v skupine")
plt.show()

adhoc_spectral = SpectralClustering(2, affinity="precomputed")
cluster_labels = adhoc_spectral.fit_predict(adhoc_matrix)
cluster_score = normalized_mutual_info_score(cluster_labels, train_labels)

print(f"Rezultat združevanje v skupine: {cluster_score}")
```




```
Rezultat združevanje v skupine: 0.8782063702756282
```

[ ]:

Priloga 12

qTorch klasifikacija

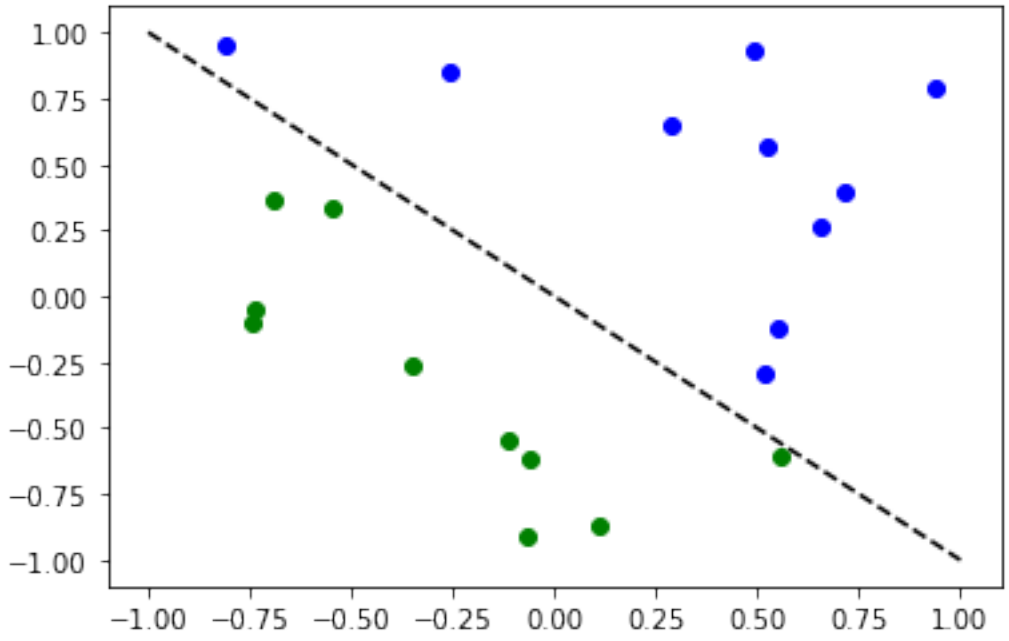
March 5, 2022

[9]:
```
import numpy as np
import matplotlib.pyplot as plt

from torch import Tensor
from torch.nn import Linear, CrossEntropyLoss, MSELoss
from torch.optim import LBFGS

from qiskit import Aer, QuantumCircuit
from qiskit.utils import QuantumInstance, algorithm_globals
from qiskit.opflow import AerPauliExpectation
from qiskit.circuit import Parameter
from qiskit.circuit.library import RealAmplitudes, ZZFeatureMap
from qiskit_machine_learning.neural_networks import CircuitQNN, TwoLayerQNN
from qiskit_machine_learning.connectors import TorchConnector

algorithm_globals.random_seed = 42
```

[10]:
```
qi = QuantumInstance(Aer.get_backend("aer_simulator_statevector"))
```

[11]:
```
num_inputs = 2
num_samples = 20

# naredimo naključne koordinate na x in y osi
X = 2 * algorithm_globals.random.random([num_samples, num_inputs]) - 1
y01 = 1 * (np.sum(X, axis=1) >= 0)
y = 2 * y01 - 1

# spremenimo v Torch tensorje
X_ = Tensor(X)
y01_ = Tensor(y01).reshape(len(y)).long()
y_ = Tensor(y).reshape(len(y), 1)

# pokažemo podatke
for x, y_target in zip(X, y):
    if y_target == 1:
        plt.plot(x[0], x[1], "bo")
    else:
```

```
        plt.plot(x[0], x[1], "go")
plt.plot([-1, 1], [1, -1], "--", color="black")
plt.show()
```



[12]:
```
# naredimo nevronsko mrežo in narišemo krog
qnn1 = TwoLayerQNN(num_qubits=num_inputs, quantum_instance=qi)
print(qnn1.operator)
initial_weights = 0.1 * (2 * algorithm_globals.random.random(qnn1.num_weights)␣
↪- 1)
model1 = TorchConnector(qnn1, initial_weights=initial_weights)
print("Initial weights: ", initial_weights)
```

```
ComposedOp([
  OperatorMeasurement(1.0 * ZZ),
  CircuitStateFn(
       ┌──────────────────────────────────┐»
  q_0: ┤0                                 ├»
       │ ZZFeatureMap(x[0],x[1]) │»
  q_1: ┤1                                 ├»
       └──────────────────────────────────┘»
  «     ┌──────────────────────────────────────────────────────────────┐
  «q_0: ┤0                                                             ├
  «     │ RealAmplitudes(θ[0],θ[1],θ[2],θ[3],θ[4],θ[5],θ[6],θ[7]) │
  «q_1: ┤1                                                             ├
  «     └──────────────────────────────────────────────────────────────┘
  )
```

```
])
Initial weights:[-0.012569620.06653564 0.04005302 -0.03752667.06645196
0.06095287
 -0.02250432 -0.04233438]
```

[13]:
```
# Testiramo s 1 qubitom
model1(X_[0, :])
```

[13]:
```
tensor([-0.3285], grad_fn=<_TorchNNFunctionBackward>)
```

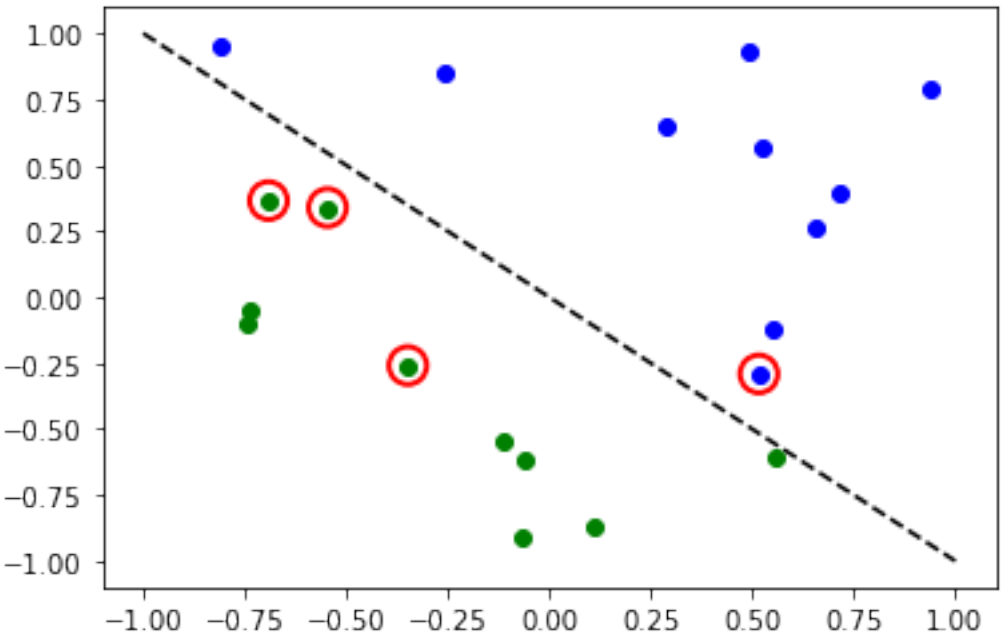
[14]:
```
optimizer = LBFGS(model1.parameters())
f_loss = MSELoss(reduction="sum")

# začnemo trening
model1.train()# nastavimo model v način treniranja

def closure():
    optimizer.zero_grad()# gradienti
    loss = f_loss(model1(X_), y#)ocenimo izgubo
    loss.backward()
    print(loss.item())
    return loss


#zaženemo optimizator
optimizer.step(closure)
```

```
25.535646438598633
22.696760177612305
20.039228439331055
19.68790626525879
19.267210006713867
19.025371551513672
18.154708862304688
17.33785629272461
19.082544326782227
17.07332420349121
16.21839141845703
14.992581367492676
14.929339408874512
14.914534568786621
14.907638549804688
14.902363777160645
14.902134895324707
14.90211009979248
14.902111053466797
```

[14]:
```
tensor(25.5356, grad_fn=<MseLossBackward0>)
```

[15]:
```
y_predict = []
for x, y_target in zip(X, y):
    output = model1(Tensor(x))
    y_predict += [np.sign(output.detach().numpy())[0]]

print("Natančnost:", sum(y_predict == y) / len(y))
# narišemo rezutate
for x, y_target, y_p in zip(X, y, y_predict):
    if y_target == 1:
        plt.plot(x[0], x[1], "bo")
    else:
        plt.plot(x[0], x[1], "go")
    if y_target != y_p:
        plt.scatter(x[0], x[1], s=200, facecolors="none", edgecolors="r",
 ↪linewidths=2)
plt.plot([-1, 1], [1, -1], "--", color="black")
plt.show()
```

```
Natančnost: 0.8
```



[ ]:

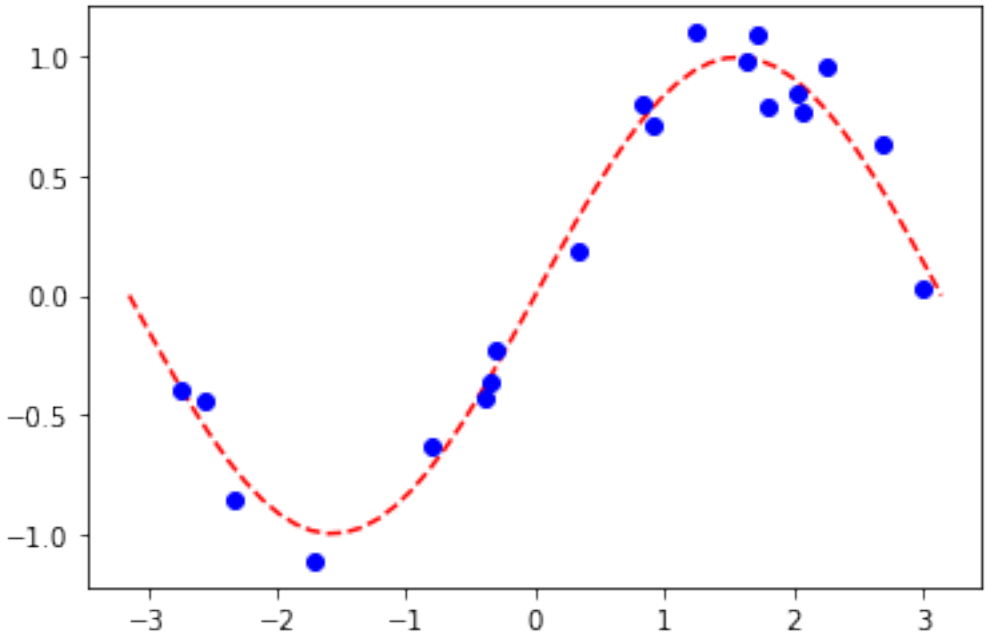
Priloga 13

qTorch regresija

March 6, 2022

[9]:
```
import numpy as np
import matplotlib.pyplot as plt

from torch import Tensor
from torch.nn import Linear, CrossEntropyLoss, MSELoss
from torch.optim import LBFGS

from qiskit import Aer, QuantumCircuit
from qiskit.utils import QuantumInstance, algorithm_globals
from qiskit.opflow import AerPauliExpectation
from qiskit.circuit import Parameter
from qiskit.circuit.library import RealAmplitudes, ZZFeatureMap
from qiskit_machine_learning.neural_networks import CircuitQNN, TwoLayerQNN
from qiskit_machine_learning.connectors import TorchConnector

algorithm_globals.random_seed = 42
# simulator
qi = QuantumInstance(Aer.get_backend("aer_simulator_statevector"))
```

[10]:
```
# določimo podatkovno bazo za treniranje in jo prikažemo s iskano funkcijo
num_samples = 20
eps = 0.2
lb, ub = -np.pi, np.pi
f = lambda x: np.sin(x)

X = (ub - lb) * algorithm_globals.random.random([num_samples, 1]) + lb
y = f(X) + eps * (2 * algorithm_globals.random.random([num_samples, 1]) - 1)
plt.plot(np.linspace(lb, ub), f(np.linspace(lb, ub)), "r--")
plt.plot(X, y, "bo")
plt.show()
```



[11]:
```
# določimo parametre modela
param_x = Parameter("x")
feature_map = QuantumCircuit(1, name="fm")
feature_map.ry(param_x, 0)

# naredimo preprosto mapo lastnosti s ansatzom
param_y = Parameter("y")
ansatz = QuantumCircuit(1, name="vf")
ansatz.ry(param_y, 0)

# definiramo in sestavimo kvantno nevronsko mrežo
qnn = TwoLayerQNN(1, feature_map, ansatz, quantum_instance=qi)
print(qnn.operator)

# nastavimo PyTorch modul (določimo uteži in naključni seed)
initial_weights = 0.1 * (2 * algorithm_globals.random.random(qnn.num_weights) -
 ↪1)
model = TorchConnector(qnn, initial_weights)
```

```
ComposedOp([
  OperatorMeasurement(1.0 * Z),
  CircuitStateFn(
     ����������������������
  q: ┤fm(x) ├┤vf(y) ├
     ����������������������
  )
```

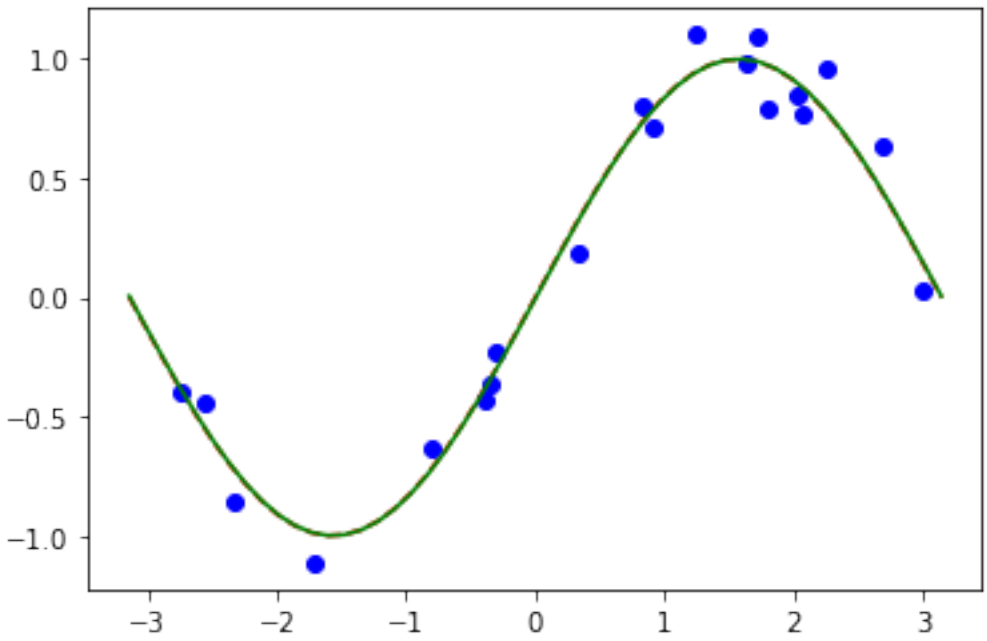
```
])
```

[12]:
```
# definiramo optimiator in funkcijo izgube
optimizer = LBFGS(model.parameters())
f_loss = MSELoss(reduction="sum")

# treniramo
model.train()

# Definiramo objektno funkcijo, kjer prikažemo izgubo
def closure():
    optimizer.zero_grad(set_to_none=True)
    loss = f_loss(model(Tensor(X)), Tensor(y))
    loss.backward()
    print(loss.item())
    return loss

# zaženemo optimizator
optimizer.step(closure)
```

```
21.778974533081055
3.426013231277466
2.017962694168091
0.2689051628112793
0.26549896597862244
0.26548153162002563
```

[12]:
```
tensor(21.7790, grad_fn=<MseLossBackward0>)
```

[14]:
```
# prikažemo iskano funkcijo in podatke
plt.plot(np.linspace(lb, ub), f(np.linspace(lb, ub)), "r--")
plt.plot(X, y, "bo")

# in našo funkcijo modela
y_ = []
for x in np.linspace(lb, ub):
    output = model(Tensor([x]))
    y_ += [output.detach().numpy()[0]]
plt.plot(np.linspace(lb, ub), y_, "g-")
plt.show()
```


1.0
0.5
0.0
−0.5
−1.0
−3
−2
−1
0
1
2
3

[ ]:

Priloga 14

qTorch pisava

March 5, 2022

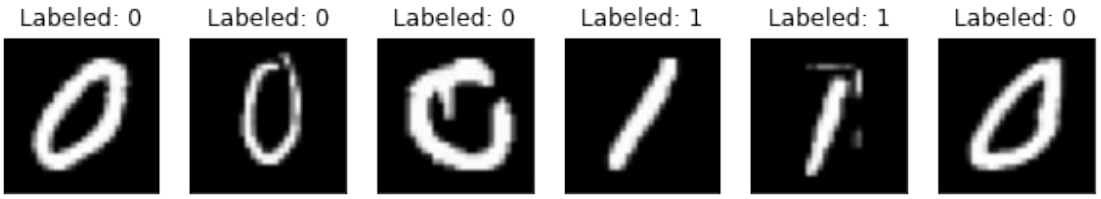
[21]:
```
import numpy as np
import matplotlib.pyplot as plt

from torch import Tensor
from torch.nn import Linear, CrossEntropyLoss, MSELoss
from torch.optim import LBFGS

from qiskit import Aer, QuantumCircuit
from qiskit.utils import QuantumInstance, algorithm_globals
from qiskit.opflow import AerPauliExpectation
from qiskit.circuit import Parameter
from qiskit.circuit.library import RealAmplitudes, ZZFeatureMap
from qiskit_machine_learning.neural_networks import CircuitQNN, TwoLayerQNN
from qiskit_machine_learning.connectors import TorchConnector

# nastavimo za naključne generatorje
algorithm_globals.random_seed = 42
from torch import cat, no_grad, manual_seed
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import torch.optim as optim
from torch.nn import (
    Module,
    Conv2d,
    Linear,
    Dropout2d,
    NLLLoss,
    MaxPool2d,
    Flatten,
    Sequential,
    ReLU,
)
import torch.nn.functional as F
import numpy as np
import torch
import torchvision
import matplotlib.pyplot as plt
```

```
from time import time
from torchvision import datasets, transforms
from torch import nn, optim
import numpy as np
import matplotlib.pyplot as plt


from torch import nn
import timeit
starttime = timeit.default_timer()
qi = QuantumInstance(Aer.get_backend("aer_simulator_statevector"))
```

[22]:
```
#izberemo podatke za treniranja
manual_seed(42)

batch_size = 1
n_samples = 100#določimo 100 vzorcev

# uporabimo torchvision funkcijo za nalaganje podatkov iz MINSTa
X_train = datasets.MNIST(
    root="./data", train=True, download=True, transform=transforms.
↪Compose([transforms.ToTensor()])
)

# pustimo samo 0 in 1 od prvotnih vzorcev
idx = np.append(
    np.where(X_train.targets == 0)[0][:n_samples], np.where(X_train.targets ==␣
↪1)[0][:n_samples]
)
X_train.data = X_train.data[idx]
X_train.targets = X_train.targets[idx]

# določimo še dataloader
train_loader = DataLoader(X_train, batch_size=batch_size, shuffle=True)
```

[23]:
```
# iz radovednosti lahko preverimo nekaj podatkov
n_samples_show = 6

data_iter = iter(train_loader)
fig, axes = plt.subplots(nrows=1, ncols=n_samples_show, figsize=(10, 3))

while n_samples_show > 0:
    images, targets = data_iter.__next__()

    axes[n_samples_show - 1].imshow(images[0, 0].numpy().squeeze(), cmap="gray")
    axes[n_samples_show - 1].set_xticks([])
    axes[n_samples_show - 1].set_yticks([])
```

```
    axes[n_samples_show - 1].set_title("Labeled: {}".format(targets[0].item()))

    n_samples_show -= 1
```



[24]:
```
# določimo podatke za testiranje
n_samples = 50

X_test = datasets.MNIST(
    root="./data", train=False, download=True, transform=transforms.
↪Compose([transforms.ToTensor()])
)

idx = np.append(
    np.where(X_test.targets == 0)[0][:n_samples], np.where(X_test.targets ==␣
↪1)[0][:n_samples]
)
X_test.data = X_test.data[idx]
X_test.targets = X_test.targets[idx]

test_loader = DataLoader(X_test, batch_size=batch_size, shuffle=True)
```

[25]:
```
# določimo še mapo lastnosti in ansatz
feature_map = ZZFeatureMap(2)
ansatz = RealAmplitudes(2, reps=1)
qnn4 = TwoLayerQNN(
    2, feature_map, ansatz, input_gradients=True,␣
↪exp_val=AerPauliExpectation(), quantum_instance=qi
)
print(qnn4.operator)
```

```
ComposedOp([
  OperatorMeasurement(1.0 * ZZ),
  CircuitStateFn(
       ����������������������������������������������������������
  q_0: ─0                    ──0                                 ─
       ┤ ZZFeatureMap(x[0],x[1]) ├┤ RealAmplitudes(θ[0],θ[1],θ[2],θ[3]) ┤
  q_1: ─1                    ──1                                 ─
       ����������������������������������������������������������
```

```
    )
])
```

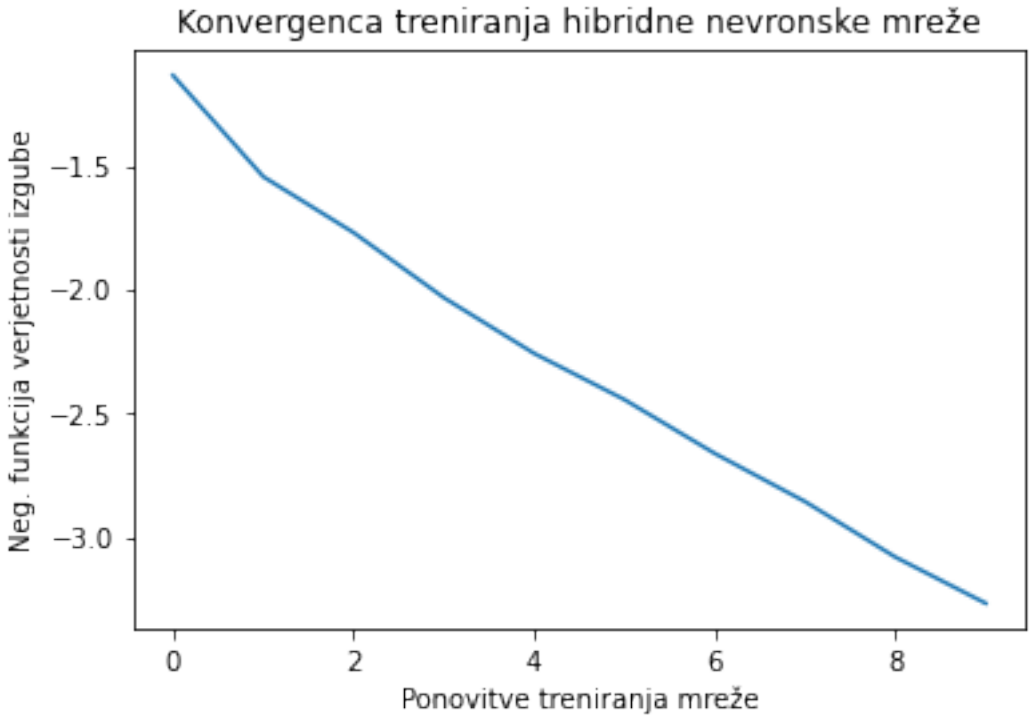
[26]:
```
# definiramo navronsko mrežo za izbrane podatke
class Net(Module):
    def __init__(self):
        super().__init__()
        self.conv1 = Conv2d(1, 2, kernel_size=5)
        self.conv2 = Conv2d(2, 16, kernel_size=5)
        self.dropout = Dropout2d()
        self.fc1 = Linear(256, 64)
        self.fc2 = Linear(64, 2) # 2-dimenzionalen input
        self.qnn = TorchConnector(qnn4) # dodamo connector in uteži
        self.fc3 = Linear(1, 1) # 1-dimenzionalen output

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        x = self.dropout(x)
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = self.qnn(x)
        x = self.fc3(x)
        return cat((x, 1 - x), -1)


model4 = Net()
```

[27]:
```
optimizer = optim.Adam(model4.parameters(), lr=0.001)
loss_func = NLLLoss()

#treniranje
epochs = 10# število epoch
loss_list = []# beležimo izgubo
model4.train()# nastavimo model za treniranje
time0 = time()

for epoch in range(epochs):
    total_loss = []
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad(set_to_none=True) # Iniliziramo gradient
        output = model4(data)
        loss = loss_func(output, target)# izračunamo izgubo
        loss.backward()
```

```
        optimizer.step()# Optimiziramo uteži
        total_loss.append(loss.item())#zabeležimo izgubo
    loss_list.append(sum(total_loss) / len(total_loss))
    print("Training [{:.0f}%]\tLoss: {:.4f}".format(100.0 * (epoch + 1) /
 ↪epochs, loss_list[-1]))
print("\nČas treniranja (sekunde) =",(time()-time0))
```

```
Training [10%]Loss: -1.1354
Training [20%]Loss: -1.5442
Training [30%]Loss: -1.7700
Training [40%]Loss: -2.0329
Training [50%]Loss: -2.2580
Training [60%]Loss: -2.4445
Training [70%]Loss: -2.6620
Training [80%]Loss: -2.8563
Training [90%]Loss: -3.0804
Training [100%] Loss: -3.2670

Čas treniranja (sekunde) = 82.92771887779236
```

[28]:
```
plt.plot(loss_list)
plt.title("Konvergenca treniranja hibridne nevronske mreže")
plt.xlabel("Ponovitve treniranja mreže")
plt.ylabel("Neg. funkcija verjetnosti izgube")
plt.show()
```

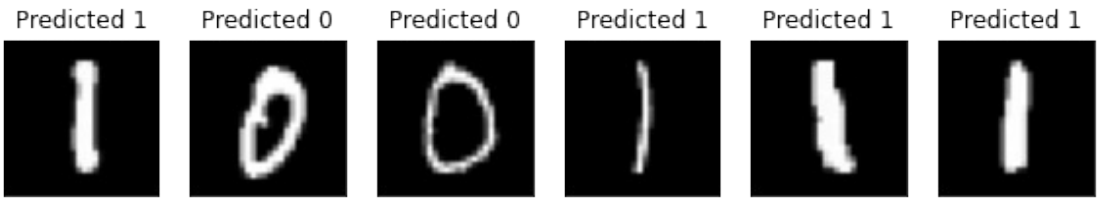


[29]:
```
model4.eval()# nastavimo model v način ocenjevanja
with no_grad():

    correct = 0
    for batch_idx, (data, target) in enumerate(test_loader):
        output = model4(data)
        if len(output.shape) == 1:
            output = output.reshape(1, *output.shape)

        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

        loss = loss_func(output, target)
        total_loss.append(loss.item())

    print(
        "Performance on test data:\n\tLoss: {:.4f}\n\tAccuracy: {:.1f}%".format(
            sum(total_loss) / len(total_loss), correct / len(test_loader) /␣
↪batch_size * 100
        )
    )
```

```
Performance on test data:
	Loss: -3.3117
	Accuracy: 100.0%
```

[30]:
```
# na koncu lahko še prikažemo nekatere podatke in njihove oznake od modela
n_samples_show = 6
count = 0
fig, axes = plt.subplots(nrows=1, ncols=n_samples_show, figsize=(10, 3))

model4.eval()
with no_grad():
    for batch_idx, (data, target) in enumerate(test_loader):
        if count == n_samples_show:
            break
        output = model4(data[0:1])
        if len(output.shape) == 1:
            output = output.reshape(1, *output.shape)

        pred = output.argmax(dim=1, keepdim=True)

        axes[count].imshow(data[0].numpy().squeeze(), cmap="gray")

        axes[count].set_xticks([])
```

```
        axes[count].set_yticks([])
        axes[count].set_title("Predicted {}".format(pred.item()))

        count += 1
```



[31]:
```
print("čas:",timeit.default_timer() - starttime)
```

```
čas: 84.23389747300098
```

Priloga 15

PyTorch pisava

March 5, 2022

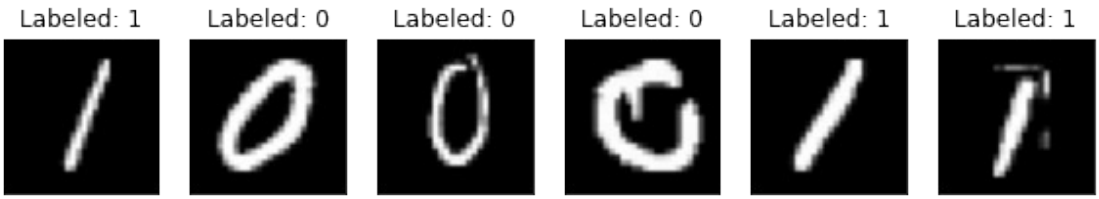
[25]:
```
import numpy as np
import torch
import torchvision
import matplotlib.pyplot as plt
from time import time
from torchvision import datasets, transforms
from torch import nn, optim
import numpy as np
import matplotlib.pyplot as plt

from torch import Tensor
from torch.nn import Linear, CrossEntropyLoss, MSELoss
from torch.optim import LBFGS

from qiskit import Aer, QuantumCircuit
from qiskit.utils import QuantumInstance, algorithm_globals
from qiskit.opflow import AerPauliExpectation
from qiskit.circuit import Parameter
from qiskit.circuit.library import RealAmplitudes, ZZFeatureMap
from qiskit_machine_learning.neural_networks import CircuitQNN, TwoLayerQNN
from qiskit_machine_learning.connectors import TorchConnector

algorithm_globals.random_seed = 42
from torch import cat, no_grad, manual_seed
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import torch.optim as optim
from torch.nn import (
    Module,
    Conv2d,
    Linear,
    Dropout2d,
    NLLLoss,
    MaxPool2d,
    Flatten,
    Sequential,
    ReLU,
```

```
)
import torch.nn.functional as F
from torch import nn
import timeit
starttime = timeit.default_timer()
```

[26]:
```
transform = transforms.Compose([transforms.ToTensor()])
manual_seed(42)
batch_size = 1
```

[27]:
```
# uvozimo podatke za treniranje iz baze
n_samples = 100
trainset = datasets.MNIST('PATH_TO_STORE_TRAINSET', download=True, train=True,␣
↪transform=transform)
trainloader = DataLoader(trainset, batch_size=1, shuffle=True)
idx = np.append(
    np.where(trainset.targets == 0)[0][:n_samples], np.where(trainset.targets␣
↪== 1)[0][:n_samples]
)
trainset.data = trainset.data[idx]
trainset.targets = trainset.targets[idx]
```

[28]:
```
n_samples_show = 6
dataiter = iter(trainloader)
images, labels = dataiter.next()

fig, axes = plt.subplots(nrows=1, ncols=n_samples_show, figsize=(10, 3))

while n_samples_show > 0:
    images, targets = dataiter.__next__()

    axes[n_samples_show - 1].imshow(images[0, 0].numpy().squeeze(), cmap="gray")
    axes[n_samples_show - 1].set_xticks([])
    axes[n_samples_show - 1].set_yticks([])
    axes[n_samples_show - 1].set_title("Labeled: {}".format(targets[0].item()))

    n_samples_show -= 1
```



[29]:
```
# uvozimo podatke za testiranje iz baze
n_samples = 50
valset = datasets.MNIST('PATH_TO_STORE_TESTSET', download=True, train=False,␣
↪transform=transform)
idx = np.append(
    np.where(valset.targets == 0)[0][:n_samples], np.where(valset.targets ==␣
↪1)[0][:n_samples]
)
valset.data = valset.data[idx]
valset.targets = valset.targets[idx]
valloader = DataLoader(valset, batch_size=1, shuffle=True)
```

[30]:
```
# definiramo nevronsko mrežo
input_size = 784
hidden_sizes = [128, 64]
output_size = 10

model = nn.Sequential(nn.Linear(input_size, hidden_sizes[0]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[0], hidden_sizes[1]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[1], output_size),
                      nn.LogSoftmax(dim=1))
print(model)
```

```
Sequential(
  (0): Linear(in_features=784, out_features=128, bias=True)
  (1): ReLU()
  (2): Linear(in_features=128, out_features=64, bias=True)
  (3): ReLU()
  (4): Linear(in_features=64, out_features=10, bias=True)
  (5): LogSoftmax(dim=1)
)
```

[31]:
```
# določimo izgubo
criterion = nn.NLLLoss()
images, labels = next(iter(trainloader))
images = images.view(images.shape[0], -1)

logps = model(images)
loss = criterion(logps, labels) #izračunamo izgubo
```

[32]:
```
# nastavimo optimizator
optimizer = optim.SGD(model.parameters(), lr=0.003, momentum=0.9)
time0 = time()
epochs = 15
for e in range(epochs):
```

```
    running_loss = 0
    for images, labels in trainloader:
        # spremenimo slike v 784 dolg vector
        images = images.view(images.shape[0], -1)

        optimizer.zero_grad()

        output = model(images)
        loss = criterion(output, labels)

        loss.backward()

        optimizer.step()

        running_loss += loss.item()
    else:
        print("Epoha {} - izguba treniranje: {}".format(e, running_loss/
 ↪len(trainloader)))
print("\nČas treniranja (sekunde) =",(time()-time0))
```

```
Epoha 0 - izguba treniranje: 0.4580219056501573
Epoha 1 - izguba treniranje: 0.0056480637716513375
Epoha 2 - izguba treniranje: 0.0012471275970744955
Epoha 3 - izguba treniranje: 0.0009031019676722707
Epoha 4 - izguba treniranje: 0.0006957294750845122
Epoha 5 - izguba treniranje: 0.0005622546981558685
Epoha 6 - izguba treniranje: 0.0004730750601656197
Epoha 7 - izguba treniranje: 0.0004065766236012891
Epoha 8 - izguba treniranje: 0.000356284624332055
Epoha 9 - izguba treniranje: 0.00031614022140196595
Epoha 10 - izguba treniranje: 0.00028463911194808844
Epoha 11 - izguba treniranje: 0.00025791467607501063
Epoha 12 - izguba treniranje: 0.00023515545099908052
Epoha 13 - izguba treniranje: 0.00021696620270372335
Epoha 14 - izguba treniranje: 0.00020053422256783194

Čas treniranja (sekunde) = 2.484940528869629
```

[33]:
```
# zdaj lahko še na enem konkretnem primeru preverimo kako bi ugibal model
images, labels = next(iter(valloader))

img = images[0].view(1, 784)
with torch.no_grad():
    logps = model(img)

ps = torch.exp(logps)
probab = list(ps.numpy()[0])
```

```
print("Ugibanje:", probab.index(max(probab)))
```

```
Ugibanje: 0
```

[34]:
```
# samo zapišemo še število testiranih slik in natančnost
correct_count, all_count = 0, 0
for images,labels in valloader:
  for i in range(len(labels)):
    img = images[i].view(1, 784)
    with torch.no_grad():
        logps = model(img)


    ps = torch.exp(logps)
    probab = list(ps.numpy()[0])
    pred_label = probab.index(max(probab))
    true_label = labels.numpy()[i]
    if(true_label == pred_label):
      correct_count += 1
    all_count += 1

print("Število testiran slik:", all_count)
print("\nNatančnost:", (correct_count/all_count))
```

```
Število testiran slik: 100

Natančnost: 1.0
```

S pomočjo Shorovega algoritma je kvantni računalnik eksponentno hitrejši pri iskanju prafaktorjev velikih polpraštevil, števil ki imajo natanko (imajo 4 delitelje, 1, sebe in še dve drugi števili). Na tem deluje današnja RSA enkripcija, ki kodira sporočila z javnim in zasebnim ključem, tako da je zasebni prafaktor javnega. Ker je iskanje prafaktorjev pri velikih praštevilih zahtevno, so sporočila zavarovana, množenje pa je z algoritmi enostavneje. Za polpraštevilo s 232 mestno decimalno vrednostjo bi potrebovali 1.000 let procesiranje na tipičnem prenosniku.

In [15]:

PRILOGA 16

```
from qiskit.aqua.algorithms import Shor
from qiskit.aqua import QuantumInstance
import numpy as np
from qiskit import QuantumCircuit, Aer, execute
from qiskit.tools.visualization import plot_histogram
#uvozimo Shor, za delovanje algoritma, Quantum Instance za zagon eksperimenta, krg, simula
```

In [16]:

```
backend = Aer.get_backend('qasm_simulator') #nastavimo simulator
eksperiment = QuantumInstance(backend, shots=1000) #nastavimo delovanje simulaotrja za eks
algoritem = Shor(N=15,a=2,quantum_instance=eksperiment) #nastavimo algoritem, tokrat bomo
```

In [17]:

```
Shor.run(algoritem)
```

Out[17]:

```
{'factors': [[3, 5]], 'total_counts': 66, 'successful_counts': 17}
```

15 smo faktorizirali pravilno, na 3 in 5. Že v teoretičnem delu naloge smo si ogledali osnovno delovanje Shorovega algoritma (stran 27). Najprej moramo ugibati število (a), katerega kvadrat je blizu števila, ki ga želimo faktorizirati. Računamo, da ima a nekatere enake prafaktorje kot N. Imamo šrevilo r, q in p. r je perioda modularne eksponentne funkcije, q je a^r/2 + 1, p pa a^r/2 - 1.

In [18]:

```
def c_amod15(a,power): #napišimo primer konkretno za 15, za katerokoli drugo število bi mo
    U = QuantumCircuit(4) #krog ima 4 qubite
    for iteration in range(power):#naredimo loop glede na število
        U.swap(2,3)
        U.swap(1,2)
        U.swap(0,1)
        for q in range(4):
            U.x(q)
    U.to_gate() #spremenimo krog v vrata
    U.name="%i^%i mod 15" %(a,power) #poimenujemo vrata
    c_U = U.control()
    return c_U
```

In [19]:

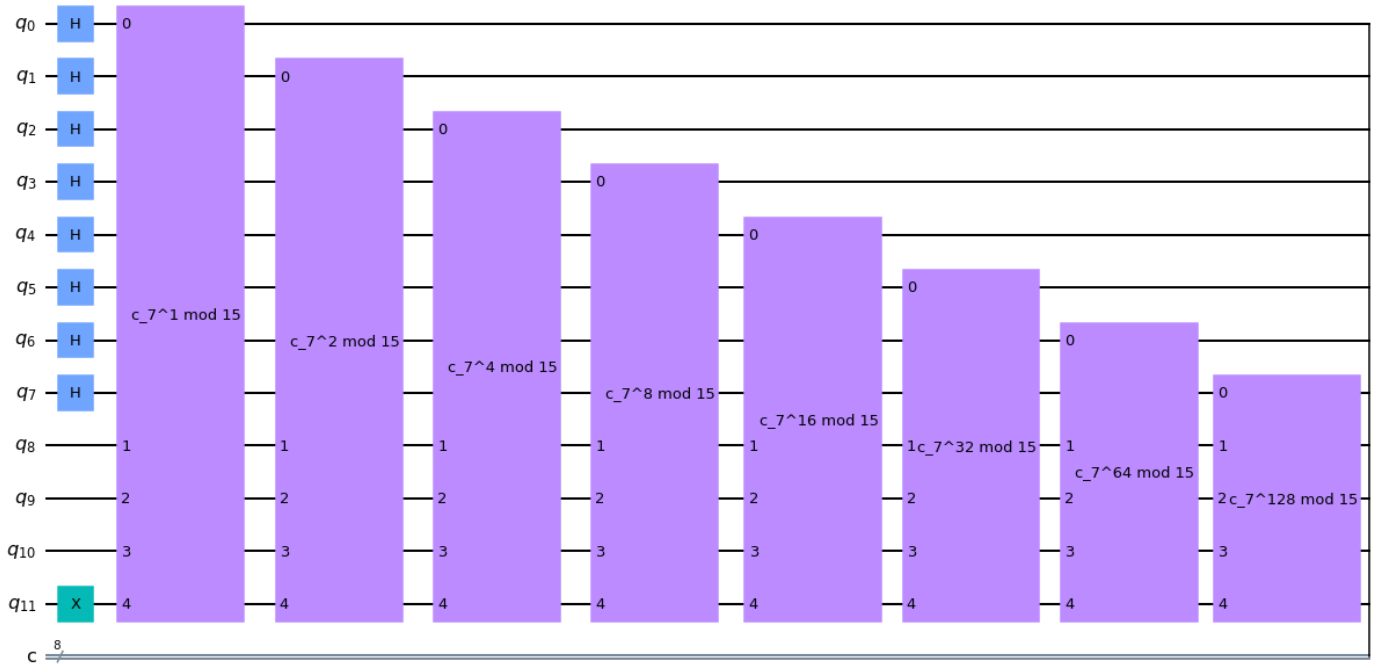
```
n_count = 8 #imamo 8 qubitov, ki bo štelo eksponent
a = 7 #ugibajoče število je 7 (a)
```

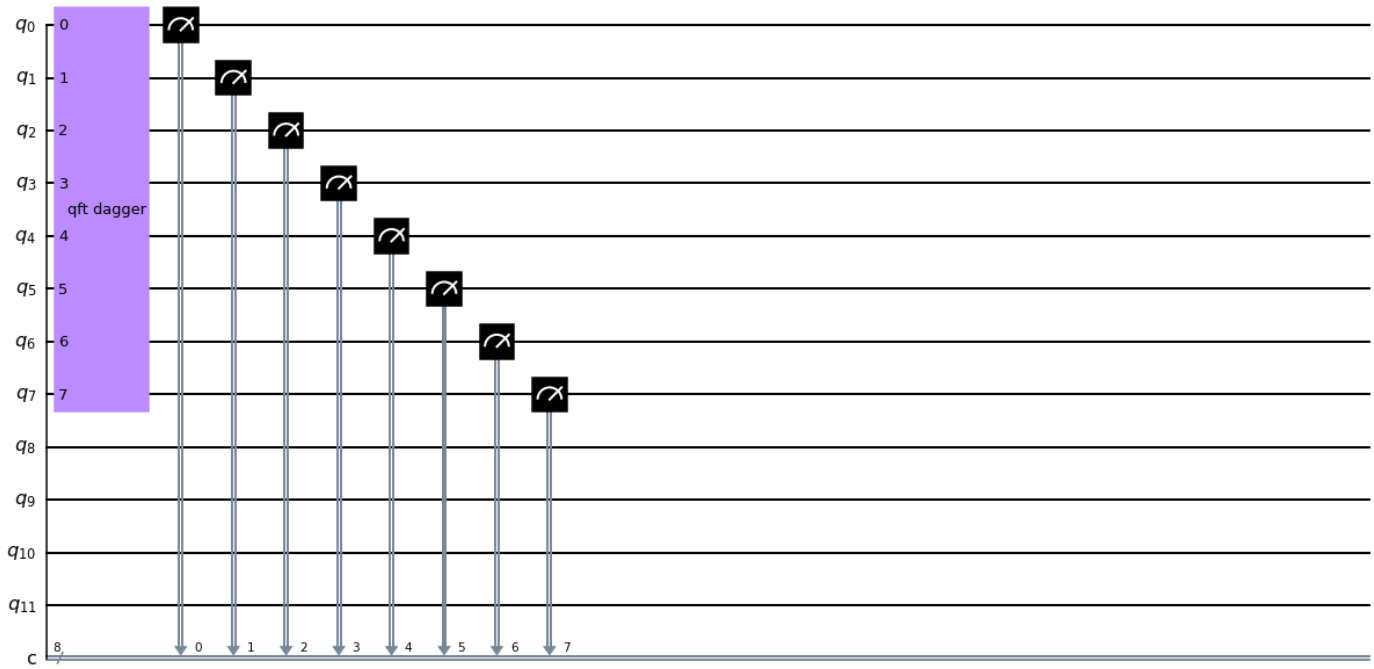
In [20]:

```
def qft_dagger(n):
    qc = QuantumCircuit(n)
    for qubit in range(n//2):
        qc.swap(qubit,n-qubit-1)
    for j in range(n):
        for m in range(j):
            qc.cu1(-np.pi/float(2**(j-m)),m,j)
        qc.h(j)
    qc.name="qft dagger"
    return qc
```

In [21]:
```
qc = QuantumCircuit(n_count + 4,n_count)

for q in range(n_count):
    qc.h(q)

qc.x(3+n_count)

for q in range(n_count):
    qc.append(c_amod15(a,2**q), [q]+[i+n_count for i in range (4)])
qc.append(qft_dagger(n_count),range(n_count))
qc.measure(range(n_count),range(n_count))
qc.draw(output='mpl')
```

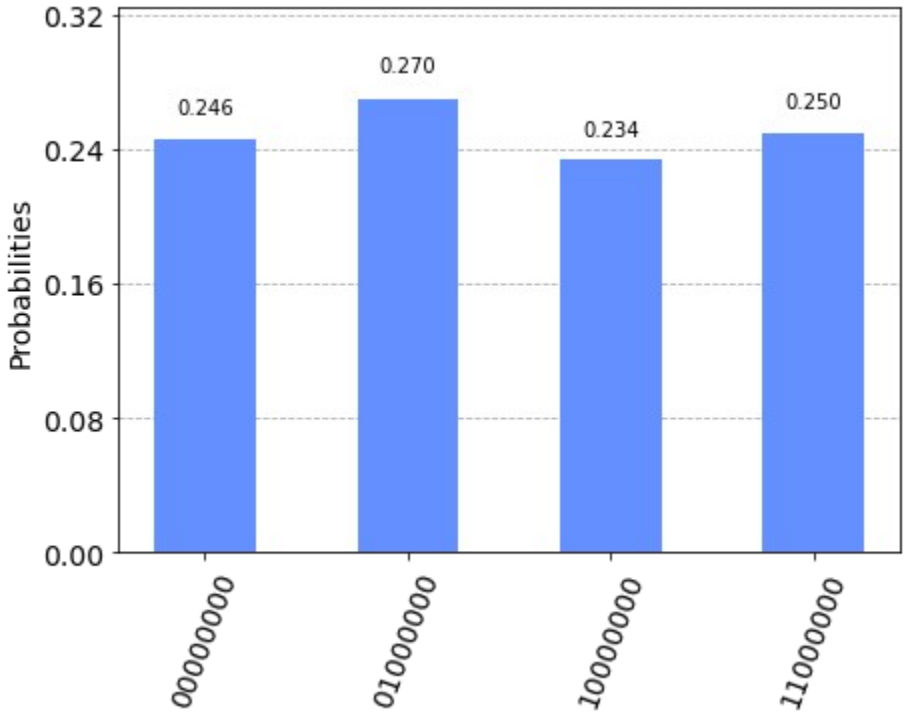
Out[21]:





In [22]:
```
backend = Aer.get_backend('qasm_simulator')
results = execute(qc, backend, shots=2048).result()
counts = results.get_counts()
plot_histogram(counts)
```

Out[22]:


0.32
0.24
0.16
0.08
0.00
Probabilities
0.246
0.270
0.234
0.250
00000000
01000000
10000000
11000000

In [ ]: